АНОТАЦІЯ

Кваліфікаційна робота присвячена розробці програмного модулю виявлення залежностей між потенційно-небезпечними дефектами початкового тексту цільових програм кібернетичного впливу. У роботі представлений аналіз теоретичних засад застосування методів теорії статистики для виявлення найбільш потенційно-вразливих ділянок коду, а також опис програмної реалізації модуля на основі залежностей між метриками інтегрованих властивостей початкового тексту цільових програм. Робота складається з вступу, 4-х розділів та списку використаних джерел. Всього 75 с, 8 рис., 1 додаток, 12 джерел.

ANNOTATION

Qualifying work is devoted to the development of a software module identify dependencies between potentially dangerous defects of the original text of target programs of cybernetic influence. The work presents the analysis of the theoretical foundations of the methods of the theory of statistics to identify the most vulnerable sections of code, and description of the software realization of the module based on the dependencies between the metrics integrated properties of the original text of target programs. The work consists of introduction, 4 chapters and bibliography. Just 75, 8 figure, 1 application, 12 sources.

Зміст

ВСТУП

Підготовка фахівців з нав'язування кібернетичного впливу та захисту вимагає напрацювання теоретичних знань і практичних навичок з використання інструментальних засобів дослідження потенційно-небезпечних дефектів вихідних текстів програм. На сьогодні існує низка засобів для здійснення аналізу та пошуку дефектів, та зазвичай вони є недостатньо інформативними та вимагають попередньої підготовки у використанні ними, тому існує необхідність у створенні більш інтерактивних, масштабованих та інформативних, що сприятиме процесу навчання. Актуальність роботи полягає у створенні програмно-технічних засобів пошуку та дослідження потенційно-небезпечних дефектів вихідних текстів програм.

Об'єкт дослідження – підготовка фахівців з аналізу та використання потенційно-небезпечних дефектів вихідних текстів програм.

Предмет дослідження – інструментальні засоби аналізу та використання потенційно-небезпечних дефектіввихідних текстів програм.

Мета роботи: Розробити програмий комплекс для відпрацювання практичних навичок фахівців з аналізу та використання потенційно-небезпечних дефектів вихідних текстів програм.

Завдання:

1. Проаналізувати напрямки підготовки фахівців з нав'язування кібернетичного впливу;
2. Побудувати програмно-технічний комплекс дослідження цільових програм на переповнення буфера;
3. Проаналізувати існуючі метрики програмного коду на предмет можливості використання їх для оцінки потенційно вразливих ділянок вихідних текстів програм;
4. Провести оцінку цінності підготовки фахівців з нав'язування кібернетичного впливу на основі переповнення буферу.

1. АНАЛІЗ МЕТОДОЛОГІЧНИХ ОСНОВ РОЗРОБКИ ЗАСОБІВ КІБЕРНЕТИЧНОГО ВПЛИВУ

1.1 Організація, принципи розробки засобів кібернетичного впливу

Виходячи зі змісту та ролі інформації у сучасному світі, американський дослідник М. Маклюен виводить цікаву тезу, що звучить так: "Істинно тотальна війна - це війна за допомогою інформації". Аналіз сучасних поглядів дозволяє вважати, що інформаційна боротьба (ІБ) у цілому є комплексом взаємопов'язаних і узгоджених за цілями, місцем і часом заходів, орієнтованих на досягнення інформаційної переваги. Вона є результатом нових інформаційних технологій. У наслідок їх застосування набули змін не тільки засоби збройної боротьби, але й стратегія, і тактика ведення сучасних воєн, з'явилися нові концепції ведення бойових дій у "інформаційному столітті", що враховують нові фактори вразливості сторін. Кібернетичний вплив – це сукупність дій, спрямованих на зміну порядку функціонування інформаційної системи.

На сьогоднішній час інформаційні технології впроваджуються у сфери діяльності людини. Не виключенням є військові організації, від рівня надійності яких напряму залежить національна безпека країни. Однак з стрімким розвитком виникає проблема захисту від комп'ютерних атак, техніка яких також постійно розвивається. Зростання можливостей щодо несанкціонованого одержання інформації, а також зацікавлення у несанкціонованому одержанні інформації, поява додаткових каналів витоку інформації, передусім у процесі обробки інформації засобами електронно-обчислювальної техніки, використання нових методів і засобів несанкціонованого здобуття інформації значно ускладнили умови інформаційної безпеки, особливо в частині протидії технічній розвідці та попередження несанкціонованої модифікації інформації шляхом зараження її вірусами і програмними закладками різних видів і типів.

Забезпечення інформаційної безпеки в умовах, що постійно змінюються, вимагає постійного проведення:

- прикладних досліджень явищ і процесів у даній предметній області, відповідно - підготовки фахівців в даній області;
- підготовки необхідної кількості підготованих і компетентних фахівців.
- збільшення чисельності фахівців, оскільки їх кількість не задовольняє існуючим потребам;
- вдосконалення навчального процесу з метою підготовки висококваліфікованих фахівців, оскільки теорія і практика інформаційної безпеки безперервно та інтенсивно розвиваються і нові досягнення повинні якнайшвидше знайти відображення у навчальних планах і програмах;

Наслідком цього процесу і стала поява певної модифікації та тенденції у системі підготовки та підвищення кваліфікації з інформаційної безпеки. Отже, виникають нові пріоритети підготовки фахівців в даній області:

1. Підготовка та перепідготовка фахівців, здатних ефективно вирішувати сучасні задачі ІБ в Україні;
2. Збільшення чисельності фахівців, які проходять підготовку та перепідготовку за напрямком інформаційної безпеки;
3. Об'єднання зусиль провідних освітніх, наукових колективів та адміністративних органів для вирішення практичних проблем ІБ;
4. Створення та постійний розвиток наукових шкіл в області ІБ;
5. Створення умов для забезпечення режиму ІБ держави в цілому, регіонів, підприємств та окремих громадян.

Для підготовки фахівців ІБ можна виділити наступні підходи:

- при збереженні однакових сфер та об'єктів професійної діяльності повинні бути зазначені відмінності за видами професійної діяльності, оскільки вони впливають на характер знань та вмінь фахівця;
- повинен бути визначений склад базових дисциплін, які, як правило, необхідні для кожної спеціальності;

- необхідно посилити і диференціювати загальнопрофесійну підготовку фахівців за кожною спеціалізацією;

- склад і зміст спеціальних дисциплін за кожною спеціалізацією повинні охоплювати інформацію, що складає всі види таємниці, розкривати всі види, методи, засоби та технологію ЗІ, однак, при цьому необхідно враховувати специфіку професійної діяльності випускника.

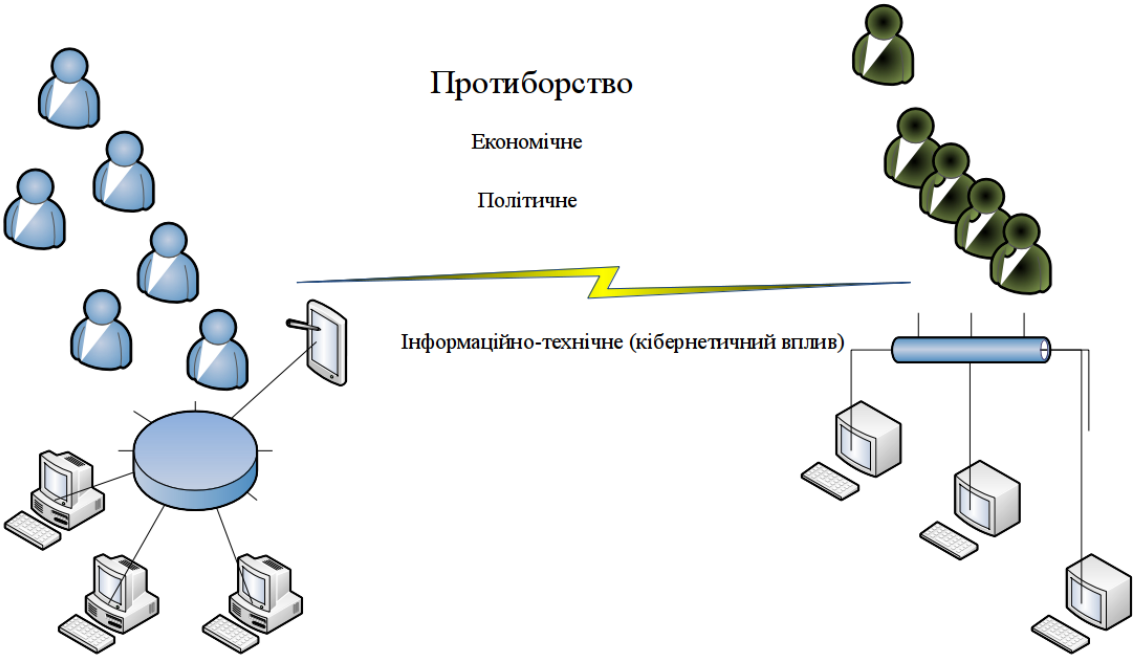

Рис. 1.1: схема кібернетичного впливу

Одним із підходів до підготовки військових фахівців, які здатні підтримувати безпеку у кібернетичному просторі – комунікаційному просторі, який охоплює комп'ютерні мережі та електронні пристрої, що використовуються для збереження, обробки та обміну інформацією є підготовка фахівців, що починається з формуваннння бази до кваліфікаційного рівня бакалавр, а саме проходження циклів підготовки. Підготовка фахівців з інформаційної безпеки (Рис 1.1)

починається у циклі професійної і практичної підготовки, де на вивчаємих дисциплінах майбутніми фахівцями вивчаються матеріали, для формування загальних знань з комп'ютерних наук, вивчаються мови програмування, технології створення програмних продуктів, архітектура комп'ютерних та операційних систем. Але щоб стати фахівцем, потрібна додаткова підготовка яка

отримується після захисту кваліфікаційної роботи, майбутні фахівці розподіляються за спеціалізаціями підготовки, фахівці формуються відповідно до спеціалізації.

На етапі підготовки згідно спеціалізації вивчаються такі дисципліни як:

- Експлуатація та бойове застосування програмних засобів інформаційної боротьби в комп'ютеризованих системах та мережах спеціального призначення;
- Методологічні основи інформаційної боротьби в комп'ютеризованих системах та мережах спеціального призначення;
- Технології побудови програмних засобів інформаційної боротьби та ін.

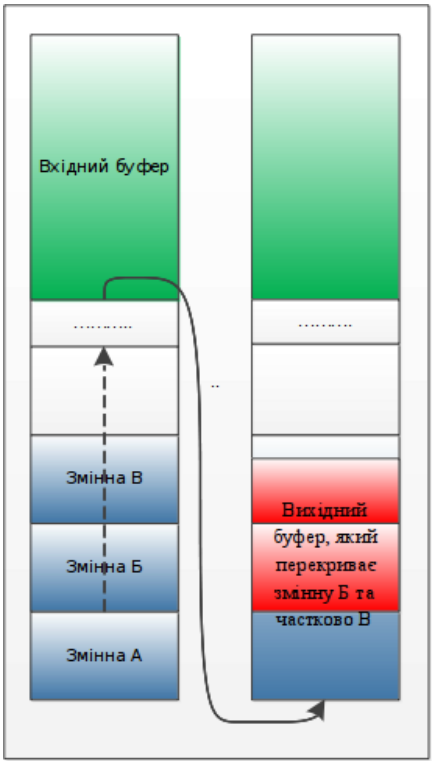

Рис. 1.2: схема переповнення буфера

На етапі підготовки згідно спеціалізації потрібно приділяти багато уваги практичній підготовці та інтерактивному навчанню. Для цього пропонується підхід до створення тренувального середовища із завідомо впровадженими вразливостями для нав'язування тестових впливів. Це середовище може слугувати матеріалом для вивчення вищеназваних дисциплін, як для відпрацювання технології побудови програмних засобів інформаційної боротьби, для

відпрацювання методології впливу на програмні продукти, а також використання програм для сканування вразливостей.

Необхідно виділити найбільш небезпечні вразливості для створення такої системи, кожна з уразливостей потребує детального вивчення, це окрема спеціалізація, яка рамках підготовки фахівця з інформаційної безпеки потребує глибокого дослідження. Досить розповсюдженим видом комп'ютерних атак на інформаційні системи є атака на переповнення буфера. Переповнення буфера було та залишається дуже важливою проблемою в аспекті безпеки програмного забезпечення. Переповнення буфера (англ. buffer overflow або англ. buffer overrun), це явище, при якому програма, під час запису даних в буфер, перезаписує дані за межами буфера. (Рис. 1.2)

Це може викликати несподівану поведінку, включно з помилками доступу до даних, невірними результатами, збоєм програми або дірою в системі безпеці. Переповнення буфера може бути викликане недостатньою перевіркою вхідних даних. Воно є базою для багатьох уразливостей в програмних продуктах і може бути злонамірено використане.

Проблема переповнення буфера з роками тільки ускладнювалася, з'являлися типи атак, в результаті були розроблені принципово нові атаки на переповнення буфера. Оскільки значна частка програмного забезпечення створюється на мові C/C++, в якій немає вбудованих засобів обробки рядків - а саме контролю розподілу пам'яті, тому вони використовують небезпечний програмний код, що не перевіряє довжину буфера, у який записуються зовнішні дані, отримані ззовні, внаслідок чого можливість перезапису інших даних програми, включаючи код, що дозволяє змінити виконання програми, незалежно коду. Атаки, в основному, здійснюються на програмні застосування, виконуються в привілейованому режимі, що дозволяє підняти рівень привілеїв для виконання шкідливого коду. Зробивши аналіз найбільш поширених прийомів техніки переповнення буферу можна зробити висновок що фахівцям в даній предметній області необхідно мати глибокі знання в таких дисциплінах як:

- операційні системи(знання архітектури ОС Linux/Unix-like/WinX);
- системне, мережеве програмування (низькорівневе);
- теорії побудови компіляторів / інтерпретаторів – теорія кінцевих авто-

матів;

- практичні навички використання засобів відлагодження та дослідження програмних продуктів;

Типові ситуації використання переповнення буфера.

Техніки використання уразливості через переповнення буфера залежать від архітектури, операційної системи і ділянки пам'яті . Існують наступні види переповнення буфера:

- переповнення у стекові (stack smashing - зрив стеку), який полягає у перезаписі адреси повернення з вразливої функції, що призводить до виконання коду(існуючого або підготовленого зловмисником) за адресом, вказаним атакуючим;

- переповнення в сегментах даних та динамічних областях (DATA, BSS, HEAP overflow), яке являє собою корекцію набору даних, керуючих алгоритмом програми, а також вказівників на функції, класи та управління.

Технічно освідомлений і злонамірений користувач може використати стекове переповнення буфера так:

1. Для перезапису локальної змінної, змінивши тим самим перебіг програми на більш вигідний для нападника.

2. Для перезапису адреси повернення в стековому кадрі. Коли буде виконане повернення з функції, виконання програми відновиться за адресою, вказаною нападником (зазвичай це адреса буфера поля вводу). Такий спосіб найбільш розповсюджений в архітектурах, де стек росте донизу (наприклад в архітектурі x86).

3. Для перезаписів вказівника на функцію, або обробника винятків, що будуть виконані згодом.

Два основних типи даних, які дозволяють здійснити переповнення буфера в цих мовах — рядки і масиви. Таким чином, використання бібліотек для рядків і спискових структур даних, які були розроблені для запобігання і/або

виявлення переповнень буфера, дозволить уникнути уразливостей. Шелкод (Корисне навантаження) (англ. *shellcode*, код запуску оболонки) - це двійковий виконуваний код, який зазвичай передає управління командному процесору, наприклад '*/ bin / sh*' *Unix Shell*, *command.com* в *MS-DOS* і *cmd.exe* в операційних системах *Microsoft Windows*. Шелкод може бути використаний як корисне навантаження експлойту, що забезпечує зловмисникові доступ до командного інтерпретатра (англ.*shell*) в комп'ютерній системі.

При експлуатації віддаленої уразливості шелкод може відкривати заздалегідь заданий порт *TCP* уразливого комп'ютера, через який буде здійснюватися подальший доступ до командної оболонці, такий код називається прив'язує до порту (англ. *port binding shellcode*). Якщо шелкод здійснює підключення до порту комп'ютера атакуючого, що проводиться з метою обходу брандмауера або *NAT*, то такий код називається зворотної оболонкою (англ. *reverse shell shellcode*). Шелкод зазвичай впроваджується в пам'ять експлуатованої програми, після чого на нього передається управління шляхом переповнення стека, або при переповненні буфера в купі, або використовуючи атаки форматної рядка. Передача управління шелкоду здійснюється перезаписом адреси повернення в стеку адресою впровадженого шелкоду, перезаписом адрес викликаються функцій або зміною обробників переривань. Результатом цього є виконання шелкоду, який відкриває командний рядок для використання зломщиком.

Розрізняють два типи шелкоду :

- локальний (впровадження та виконання можливе тільки за безпосереднього доступу зловмисника до КС)
- віддалений (атака з віддаленої машини).

Локальний шелкод використовується зловмисником, коли він обмежений в правах комп'ютерної системи, але може використати вразливість в програмному забезпеченні даної системи з підняттям привілеїв до того рівня що і цільовий процес.

Віддалений шелкод використовується хакерами для захоплення контролю над процесом на віддаленій машині в локальній мережі чи в інтернеті. У разі успішного його виконання, хакер матиме можливість отримати доступ над машиною по мережі. Віддалені шелкоди переважно використову-

ють *TCP/IP*-зєднання для забезпечення атакуючого доступом до командної оболонки. Такий шелкод може бути класифікований по тому, як він встановлює з'єднання.: якщо код може встановити з'єднання, його називають "*reverse shell*" або "*connect-back shellcode*" , так як він з'єднується з машиною атакуючого. З іншого боку, якщо зловмисник потребує встановити з'єднання, то такий шелкод називають "*bindshell*", так як він відкриває і прослуховує порт, по якому зловмисник може під'єднатися і отримати контроль над системою. Є третій тип, менш використовуваний, має назву "*socket-reuse shellcode*". Цей тип шелкоду іноді використовується, коли експлойт встановлює з'єднання з вразливим процесом, який до його запуску не закрив з'єднання. Тому він може використати з'єднання для комунікації з хакером для віддаленого управління системою. Цей тип шелкоду є складним в реалізації так як він повинен виконати пошук вразливого процесу, який має відкрите з'єднання.

1.2 Дослідження початкового тексту як метод виявлення потенційно-небезпечних дефектів програм.

На сьогоднішній час існує низка засобів для аналізу програм як у вигляді вихідних текстів так і у двійковому коді. Для виявлення вразливостей захисту в програмах застосовують такі інструментальні засоби:

- динамічні відлагоджувачі. Інструменти, які дозволяють виробляти налагодження програми в процесі її виконання;
- статичні аналізатори (статичні відлагоджувачі).

Інструменти, які використовують інформацію, накопичену в ході статичного аналізу досліджуваної програми.

Статичні аналізатори вказують на ті місця в програмі, в яких можливо знаходиться помилка. Ці підозрілі фрагменти коду можуть, як містити помилку, так і не нести ніякої небезпеки для виконання програми. Наявність вихідних кодів програми істотно спрощує пошук вразливостей. Розглянемо декілька інструментів для аналізу вихідних текстів досліджуваних програм:

Інструмент *BOON*, який на основі глибокого семантичного аналізу автоматизує процес сканування вихідних текстів на Сі в пошуках уразливих місць, здатних призводити до переповнення буферу. Він виявляє можливі дефекти, припускаючи, що деякі значення є частиною неявного типу з конкретним розміром буфера.

CQual - інструмент аналізу для виявлення помилок в Сі-програмах. Програма розширює мову Сі додатковими обумовленими користувачем специфікаторами типу. Програміст коментує свою програму з відповідними специфікаторами, і *cqual* перевіряє помилки. Неправильні анотації вказують на потенційні помилки. *Cqual* може використовуватися, щоб виявити потенційну уразливість форматною рядка.

MOPS - інструмент для пошуку вразливостей в захисті в програмах на Сі. Його призначення: динамічне коректування, що забезпечує відповідність програми на Сі статичної моделі. *MOPS* використовує модель аудиту програмного забезпечення, яка покликана допомогти з'ясувати, чи відповідає програма набору правил, визначеному для створення безпечних програм.

ITS4. Простий інструмент, який статично переглядає вихідний Сі / Сі++ - код для виявлення потенційних вразливостей захисту. Він зазначає виклики потенційно небезпечних функцій, таких, наприклад, як *strcpy / memcpy*, і виконує поверхневий семантичний аналіз, намагаючись оцінити, наскільки небезпечний такий код, а так само дає поради щодо його поліпшення.

RATS. Утиліта *RATS (Rough Auditing Tool for Security)* обробляє код, написаний на Сі / Сі++, а також може обробити ще і скрипти на *Perl, PHP і Python. RATS* переглядає вихідний текст, знаходячи потенційно небезпечні звернення до функцій. Мета цього інструменту - не остаточно знайти помилки, а забезпечити обгрунтовані висновки, спираючись на які фахівець зможе вручну виконувати перевірку коду.*RATS* використовує поєднання перевірок надійності захисту від семантичних перевірок в *ITS4* до глибокого семантичного аналізу в пошуках дефектів, здатних привести до переповнення буфера, отриманих з *MOPS*.

Flawfinder. Як і *RATS*, це статичний сканер вихідних текстів програм, написаних на *C/C++*. Виконує пошук функцій, які найчастіше використовуються некоректно, присвоює їм коефіцієнти ризику (спираючись на таку інформацію, як передані параметри) і складає список потенційно вразливих місць, впорядковуючи їх за ступенем ризику.

Всі ці інструменти схожі і використовують тільки лексичний і найпростіший синтаксичний аналіз. Тому результати, видані цими програмами, можуть містити до 100% помилкових повідомлень.

Bunch - засіб аналізу та візуалізації програм на Сі, яке будує граф залежностей, що допомагає аудитору розібратися в модульній структурі досліджуваної програми.

Frama-C - відкритий, інтегрований набір інструментів для аналізу вихідного коду на мові Сі. Набір включає *ACSL (ANSI / ISO C Specification Language)* - спеціальна мова, що дозволяє детально описувати специфікації функцій Сі, наприклад, вказати діапазон допустимих вхідних значень функції і діапазон нормальних вихідних значень.

Цей інструментарій допомагає виробляти такі дії:

- здійснювати формальну перевірку коду;
- шукати потенційні помилки виконання;

- провести аудит або рецензування коду;
- проводити реверс-інжиніринг коду для поліпшення розуміння структури програмного коду;
- генерувати формальну документацію.

Також при відсутності вихідного тексту для аналізу програмного коду можна використовувати динамічні відлагоджувачі, які також можуть допомогти виявити помилки в коді, які допущені компілятором. Найпоширенішими відлагоджувачами на даний момент є *SoftIce, OllyDebug, IDA Pro, GDB:*

SoftIce – всім відомий відлагоджувальник для ОС сімейства *Windows*, який працює в режимі ядра, що дозволяє відлагоджувати драйвера та різного роду сервіси що працюють в привілейованому режимі процесора. Працює в обхід *MS Debugging API*, що дуже ускладнює реалізацію захисту від відлагодження. *SoftICE* був спочатку розроблений компанією *NuMega*, яка включала його в пакет програм для швидкої розробки високопродуктивних драйверів під назвою *Driver Studio*, який пізніше був придбаний *Compuware*.

OllyDebug – це 32-бітний відлагоджувальник працюючий в непривілейованому режимі процесора. Він має достатньо зручний інтерфейс та корисні функції які значно полегшують процес від лагодження. В *OllyDBG* вбудований спеціальний аналізатор, який розпізнає і візуально позначає процедури, цикли, константи і рядки, звернення до функцій *API*, параметри цих функцій і т.п.

IDA Pro – це одночасно інтерактивний дизасемблер і відлагоджувальник. Він дозволяє отримати асемблерний текс, який може бути застосований для аналізу роботи програми. Варто зазначити, що вбудований відлагоджувальник доволі примітивний, працює через *MS Debugging API (в NT)* і через бібліотеку ptrace (в *UNIX*), що робить його легкорозпізнаємим для захисних механізмів. Сильною стороною цього продукту є саме дизасемблер, який на сьогодні генерує якісний вихідний текс. Окрім цього існує велика кількість плагінів під даний програмний продукт, що значно розширює його можливості. Також можливе написання плагінів на скрипкових мовах – є підтримка *Ruby, Python*.

GDB - GNU Debugger – основний відлагоджувальник під *UNIX*, орієнтований на зовсім інший тип мислення, аніж всі вищеперераховані відлаго-

джувальники. Це не просто інтерактивний відлагоджувальник – це модуль управління виконанням програм з гнучким і потужним інтерфейсом. Негативною стороною даного відлагоджувальника є відсутність аналізу захисних механізмів програм, тому у разі їх присутності в коді – процес відлагоджування стає неможливим.

Дані програмні засоби добре справляються з заявленими ними завданнями, але враховуючи тенденції теперішніх технологій та стрімкий ріст розмірів як вихідних текстів програм так і їх двійкових образів, завдання швидкого аналізу дещо ускладюється насамперед тим, що результати аналізу містять надмірну кількість інформації, яку також потрібно аналізувати. Тому виникає потреба в більш загальному аналізі характеристик коду, який дасть якісну оцінку окремих частин досліджуванної програми, що дозволить диференціювати задачу аналізу та правильно його організувати.

Завдання оцінки інтегрованих властивостей програм в певній мірі вирішують так звані метрики коду.

Метрика якості програм - система вимірювань якості програм . Ці виміри можуть проводитися на рівні критеріїв якості програм або на рівні окремих характеристик якості. У першому випадку система вимірювань дозволяє безпосередньо порівнювати програми за якістю. При цьому самі виміри не можуть бути проведені без суб'єктивних оцінок властивостей програм. У другому випадку вимірювання характеристик можна виконати об'єктивно і достовірно, але оцінка якості ПЗ в цілому буде пов'язана з суб'єктивною інтерпретацією одержуваних оцінок.

Висновки

Отже, в зрізі теперішніх тенденцій інформаційних технологій - а саме стрімкого росту розмірів як вихідних текстів програм так і їх двійкових образів виникає потреба в більш загальному аналізі характеристик коду, який дасть якісну оцінку окремих частин досліджуванної програми, що дозволить диференціювати задачу аналізу та правильно його організувати.

2. ТЕОРЕТИЧНІ ЗАСАДИ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ПОШУКУ (ДОСЛІДЖЕННЯ) ПОТЕНЦІЙНО-НЕБЕЗПЕЧНИХ ДЕФЕКТІВ ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЇ ЦІЛЬОВИХ ПРОГРАМ

2.1 Інтегровані властивості цільової програми

На відміну від більшості галузей матеріального виробництва, в питаннях проектів створення ПЗ неприпустимі прості підходи, засновані на множенні трудомісткості на середню продуктивність праці. Це викликано, насамперед, тим, що економічні показники проекту нелінійно залежать від обсягу робіт, а при обчисленні трудомісткості допускається велика похибка.

Тому для вирішення цього завдання використовуються комплексні і досить складні методики, які вимагають високої відповідальності в застосуванні і певного часу на адаптацію (настройку коефіцієнтів).

Сучасні комплексні системи оцінки характеристик проектів створення ПЗ можуть бути використані для вирішення наступних завдань:

- попередня, постійна і підсумкова оцінка економічних параметрів проекту: трудомісткість, тривалість, вартість;
- оцінка ризиків по проекту: ризик порушення строків та невиконання проекту, ризик збільшення трудомісткості на етапах налагодження та супроводження проекту та пр;
- прийняття оперативних управлінських рішень - на основі відстеження певних метрик проекту можна своєчасно попередити виникнення небажаних ситуацій і усунути наслідки непродуманих проектних рішень.

Для вимірювання характеристик і критеріїв якості використовують метрики.

Метрика якості програм - система вимірювань якості програм . Ці виміри можуть проводитися на рівні критеріїв якості програм або на рівні окремих характеристик якості. У першому випадку система вимірювань дозволяє безпосередньо порівнювати програми за якістю. При цьому самі виміри

не можуть бути проведені без суб'єктивних оцінок властивостей програм. У другому випадку вимірювання характеристик можна виконати об'єктивно і достовірно, але оцінка якості ПЗ в цілому буде пов'язана з суб'єктивною інтерпретацією одержуваних оцінок.

У дослідженні метрик ПО розрізняють два основних напрямки:

- Пошук метрик, що характеризують найбільш специфічні властивості програм, тобто метрик оцінки самого ПЗ;
- Використання метрик для оцінки технічних характеристик і факторів розробки програм, тобто метрик оцінки умов розробки програм.

По виду інформації, одержуваної при оцінці якості ПО метрики можна розбити на три групи:

- Метрики, що оцінюють відхилення від норми характеристик вихідних проектних матеріалів . Вони встановлюють повноту заданих технічних характеристик вихідного коду.
- Метрики, що дозволяють прогнозувати якість розроблюваного ПЗ. Вони задані на безлічі можливих варіантів рішень поставленого завдання і їх реалізації і визначають якість ПЗ, яке буде досягнуто в результаті.
- Метрики, за якими приймається рішення про відповідність кінцевого ПО заданим вимогам . Вони дозволяють оцінити відповідність розробки заданим вимогам .

Основні намрямки застосуваня метрик

В даний час у світовій практиці використовується кілька сотень метрик програм. Існуючі якісні оцінки програм можна згрупувати по шести напрямках :

- Оцінки топологічної та інформаційної складності програм;
- Оцінки надійності програмних систем, що дозволяють прогнозувати отказовие ситуації ;

- Оцінки продуктивності ПО і підвищення його ефективності шляхом виявлення помилок проектування ;
- Оцінки рівня мовних засобів і їх застосування;
- Оцінки труднощі сприйняття і розуміння програмних текстів, орієнтовані на психологічні фактори, суттєві для супроводу і модифікації програм;
- Оцінки продуктивності праці програмістів для прогнозування термінів розробки програм і планування робіт зі створення програмних комплексів.

Метричні шкали

Залежно від характеристик і особливостей застосовуваних метрик їм ставляться у відповідність різні вимірювальні шкали.

Номінальною шкалою відповідають метрики, що класифікують програми на типи за ознакою наявності або відсутності деякої характеристики без урахування градацій.

Порядкової шкалою відповідають метрики, що дозволяють ранжувати деякий характеристики шляхом порівняння з опорними значеннями, тобто вимір за цією шкалою фактично визначає взаємне положення конкретних програм.

Інтервального шкалою відповідають метрики, які показують не тільки відносне положення програм, але і те, як далеко вони відстоять один від одного.

Відносній шкалі відповідають метрики, що дозволяють не тільки розташувати програми певним чином і оцінити їх положення відносно один одного, а й визначити, як далеко оцінки відстоять від кордону, починаючи з якої характеристика може бути виміряна.

Метрика складності: При оцінці складності програм, як правило, виділяють три основні групи метрик:

- Метрики розміру програм
- Метрики складності потоку управління програм

- Метрики складності потоку даних програм

Метрика розміру програм.

Оцінки першої групи найбільш прості і, очевидно, тому отримали широке поширення. Традиційною характеристикою розміру програм є кількість рядків вихідного тексту. Під рядком розуміється будь-який оператор програми, оскільки саме оператор, а не окремо взята рядок є тим інтелектуальним " квантом " програми, спираючись на який можна будувати метрики складності її створення. Безпосереднє вимірювання розміру програми, незважаючи на свою простоту, дає хороші результати. Звичайно, оцінка розміру програми недостатня для прийняття рішення про її складності, але цілком застосовна для класифікації програм, істотно різняться обсягами. При зменшенні відмінностей в обсязі програм на перший план висуваються оцінки інших факторів, що впливають на складність . Таким чином, оцінка розміру програми є оцінка за номінальною шкалою, на основі якої визначаються тільки категорії програм без уточнення оцінки для кожної категорії .

До групи оцінок розміру програм можна віднести також і метрику Холстеда .

Метрика Холстеда. Основу метрики Холстеда складають чотири вимірюваних характеристики програми : n1 - число унікальних операторів програми, включаючи символи - роздільники, імена процедур і знаки операцій (словник операторів) ; n2 - число унікальних операндів програми (словник операндів) ; N1 - загальне число операторів в програмі ; N2 - загальне число операндів в програмі. Спираючись на ці характеристики, одержувані безпосередньо при аналізі вихідних текстів програм, М. Холстед вводить такі оцінки: словник програми n1 = n1 + n2, довжину програми N = N1 + N2, (1) обсяг програми V = N * log2 (n) (біт).

Метрика складності потоку управління програми. Друга найбільш представницька група оцінок складності програм - метрики складності потоку керування програм. Як правило, за допомогою цих оцінок оперують або щільністю керуючих переходів усередині програм, або взаємозв'язками цих переходів .

І в тому і в іншому випадку стало традиційним уявлення програм у вигляді керуючого орієнтованого графа G = (V, E), де V - вершини, відповідні операторам, а E - дуги, відповідні переходам .

Метрика Маккейба. Вперше графічне представлення програм було запропоновано МакКейб . Основний метрикою складності він пропонує вважати цикломатическая складність графа програми, або, як її ще називають, цикломатичне число МакКейб, що характеризує трудомісткість тестування програми . Для обчислення цикломатическая числа МакКейб Z (G) застосовується формула Z(G) = e - v +2 p, де e - число дуг орієнтованого графа G ; v - число вершин ; p - число компонентів зв'язності графа. Число компонентів зв'язності графа можна розглядати як кількість дуг, які необхідно додати для перетворення графа в сильно зв'язний . Сильний зв'язковим називається граф, будь-які дві вершини якого взаємно досяжні. Для графів коректних програм, тобто графів, що не мають недосяжних від точки входу ділянок і "висячих" точок входу і виходу, сильно зв'язний граф, як правило, виходить шляхом замикання дугою вершини, що позначає кінець програми, на вершину, що позначає точку входу в цю програму. По суті Z (G) визначає число лінійно незалежних контурів у Сильно зв'язкового графі . Інакше кажучи, цикломатичне число МакКейб показує необхідну кількість проходів для покриття всіх контурів сильно зв'язного графа або кількість тестових прогонів програми, необхідних для вичерпного тестування за критерієм " працює кожна гілка" . Для зображеної програми, цикломатичне число при e = 10, v = 8, p = 1 визначиться як Z (G) = 10-8 +2 = 4 .

Цикломатичне число залежить тільки від кількості предикатів, складність яких при цьому не враховується. Обидва оператора припускають єдине розгалуження і можуть бути представлені одним і тим же графом. Очевидно, цикломатичне число буде для обох операторів однаковим, що не відображає складності предикатів, що досить істотно при оцінці програм.

Метрика Маєрса. Виходячи з цього Г. Майерс запропонував розширення цієї метрики . Суть підходу Г. Майерса полягає в представленні метрики складності програм у вигляді інтервалу [Z (G), Z (G) + h] . Для простого предиката h = 0, а для n-місних предикатів h = n - 1 . Таким чином, перший оператору відповідає інтервал [2,2], а другий - [2,6] . По ідеї така метрика дозволяє розрізняти програми, представлені однаковими графами. На жаль, інформація про результати використання цього методу відсутня, тому нічого не можна сказати про його застосовності .

Метрика підрахунку точок перетину. Розглянемо метрику складності про-

грам, що отримала назву " підрахунок точок перетину ", авторами якої є М.Вудвард, М.Хенель і Д.Хідлі. Метрика орієнтована на аналіз програм, при створенні яких використовувалося неструктурні кодування на таких мовах, як мова асемблера і фортран. У графі програми, де кожному оператору відповідає вершина, тобто не виключені лінійні ділянки, при передачі управління від вершини a до b номер оператора a дорівнює min (a, b), а номер оператора b - max (a, b) . Точка перетину дуг з'являється, якщо min (a, b) < min (p, q) < max (a, b) & max (p, q) > max (a, b) | min (a, b) < max (p, q) < max (a, b) & min (p, q) < min (a, b) . Іншими словами, точка перетину дуг виникає у разі виходу управління за межі пари вершин (a, b) (рис. 3) . Кількість точок перетину дуг графа програми дає характеристики не структурованості програми .

Метрика Джилба. Однією з найбільш простих, але, як показує практика, досить ефективних оцінок складності програм є метрика Т. Джилба, в якій логічна складність програми визначається як насиченість програми виразами типу IF- THEN - ELSE . При цьому вводяться дві характеристики : CL - абсолютна складність програми, що характеризується кількістю операторів умови; cl - відносна складність програми, що характеризується насиченістю програми операторами умови, тобто cl визначається як відношення CL до загального числа операторів. Використовуючи метрику Джилба, ми доповнили її ще однією складовою, а саме характеристикою максимального рівня вкладеності оператора CLI, що дозволило не тільки уточнити аналіз по операторам типу IF- THEN - ELSE, але й успішно застосувати метрику Джилба до аналізу циклічних конструкцій.

Метрика граничних значень. Великий інтерес представляє оцінка складності програм за методом граничних значень . Введемо кілька додаткових понять, пов'язаних з графом програми. Нехай G = (V, E) - орієнтований граф програми з єдиною початковою і єдиною кінцевою вершинами. У цьому графі число входять вершин у дуг називається негативною ступенем вершини, а число що виходять з вершини дуг - позитивної ступенем вершини . Тоді набір вершин графа можна розбити на дві групи: вершини, у яких позитивна ступінь <= 1 ; вершини, у яких позитивна ступінь > = 2 . Вершини першої групи назвемо приймаючими вершинами, а вершини другої групи - вершинами відбору. Для отримання оцінки за методом граничних значень необхідно

розбити граф G на максимальне число подграфов G ‘, що задовольняють таким умовам: вхід в подграф здійснюється тільки через вершину відбору; кожен подграф включає вершину (звану надалі нижньою межею подграфа), в яку можна потрапити з будь іншої вершини подграфа . Наприклад, вершина відбору, поєднана сама з собою дугою - петлею, утворює подграф. Число вершин, що утворюють такий подграф, одно скоригованої складності вершини відбору. Кожна приймаюча вершина має скориговану складність, рівну 1, крім кінцевої вершини, скоригована складність якої дорівнює 0 . Скориговані складності всіх вершин графа G підсумовуються, утворюючи абсолютну граничну складність програми . Після цього визначається відносна гранична складність програми :

S0 = 1 - (v - 1) / Sa, де S0 - відносна гранична складність програми ; Sa - абсолютна гранична складність програми ; v - загальне число вершин графа програми .

Метрика складності потоку даних. Інша група метрик складності програм - метрики складності потоку даних, тобто використання, конфігурації і розміщення даних в програмах.

Метрика обігу до глобальних змінних. Розглянемо метрику, що зв’язує складність програм із зверненнями до глобальних змінних . Пара ” модуль - глобальна змінна” позначається як (p, r), де p - модуль, що має доступ до глобальної змінної r . Залежно від наявності в програмі реального обігу до змінної r формуються два типи пар ” модуль - глобальна змінна” : фактичні і можливі . Можливе звернення до r за допомогою p показує, що область існування r включає в себе p.

Характеристика Aup говорить про те, скільки разів модулі Up дійсно отримували доступ до глобальних змінних, а число Pup - скільки разів вони могли б отримати доступ.

Відношення числа фактичних звернень до можливих визначається

$$Rup = Aup/Pup \tag{2.1}$$

Ця формула показує наближену ймовірність посилання довільного модуля на довільну глобальну змінну. Очевидно, чим вище ця вірогідність, тим вище ймовірність несанкціонованої зміни якої-небудь змінної, що може істо-

тно ускладнити роботи, пов"язані з модифікацією програми. На жаль, поки не можна сказати, наскільки зручний і точний цей метод на практиці, так як немає відповідних статистичних даних.

Метрика Спен. Визначення Спен грунтується на локалізації звернень до даних всередині кожної програмної секції. Спен - це число тверджень, які містять даний ідентифікатор, між його першим і останнім появою в тексті програми . Отже, ідентифікатор, що з'явився n раз, має Спен, рівний n - 1 . При великому Спен ускладнюється тестування і налагодження.

Метрика Чепіна. Суть методу полягає в оцінці інформаційної міцності окремо взятого програмного модуля за допомогою аналізу характеру використання змінних зі списку вводу-виводу. Всі безліч змінних, складових список введення-виведення, розбивається на 4 функціональні групи:

- P - що вводяться змінні для розрахунків та для забезпечення виведення. Прикладом може служити використовувана в програмах лексичного аналізатора змінна, що містить рядок вихідного тексту програми, тобто сама змінна не модифікується, а лише містить вихідну інформацію .
- M - модифікуються, або створювані всередині програми змінні.
- C - змінні, що беруть участь в управлінні роботою програмного модуля (керуючі змінні).
- T - які не використовуються в програмі (" паразитні ") змінні. Оскільки кожна змінна може виконувати одночасно кілька функцій, необхідно враховувати її в кожній відповідній функціональній групі .

Далі вводиться значення метрики Чепіна :

Q = a1 * P + a2 * M + a3 * C + a4 * T, (4) де a1, a2, a3, a4 - вагові коефіцієнти .

Вагові коефіцієнти у виразі (4) використані для відбиття різного впливу на складність програми кожної функціональної групи . На думку автора метрики, найбільшу вагу, що дорівнює трьом, має функціональна група C, так як вона впливає на потік управління програми . Вагові коефіцієнти решти груп розподіляються наступним чином : a1 = 1, a2 = 2, a4 = 0.5 . Ваговий

коефіцієнт групи T НЕ дорівнює 0, оскільки " паразитні " змінні не збільшують складність потоку даних програми, але іноді ускладнюють її розуміння . З урахуванням вагових коефіцієнтів вираз (4) приймає вигляд:

Q = P + 2M + 3C + 0.5T Слід зазначити, що розглянуті метрики складності програм засновані на аналізі вихідних текстів програм і графів, що забезпечує єдиний підхід до автоматизації з розрахунку .

Метрика стилістики та зрозумілої програми

Метрика рівня коментування програм. Найбільш простий метрикою стилістики та зрозумілості програм є оцінка рівня коментування програми F : F = Nком / Nстр, де Nком - кількість коментарів у програмі ; Nстр - кількість рядків або операторів вихідного тексту .

Таким чином, метрика F відображає насиченість програми коментарями.

Виходячи з практичного досвіду прийнято вважати, що F = 0.1, тобто на кожні десять рядків програми має припадати мінімум один коментар. Як показують дослідження, коментарі розподіляються по тексту програми нерівномірно: на початку програми їх надлишок, а в середині або в кінці - недолік. Це пояснюється тим, що на початку програми, як правило, розташовані оператори опису ідентифікаторів, що вимагають більш щільного коментування. Крім того, на початку програми також розташовані шапки, що містять загальні відомості про виконавця, характері, функціональне призначення програми і т. п. Така насиченість компенсує недолік коментарів у тілі програми, і тому формула (5) недостатньо точно відображає коментування функціональної частини тексту програми. Більш вдалий варіант, коли вся програма розбивається на n рівних сегментів і для кожного з них визначається Fi:

Fi = sign (Nком / Nстр - 0.1), при цьому

n F = Сума (Fi) . i = 1 Рівень коментування програми вважається нормальним, якщо виконується умова: F = n . В іншому випадку небудь фрагмент програми доповнюється коментарями до номінального рівня .

Моделі та метрики оцінки якості ПЗ.

Сучасна програмна індустрія за півстоліття шукань накопичила значну колекцію моделей і метрик, що оцінюють окремі виробничі та експлуатаційні властивості ПЗ. Однак гонитва за їх універсальністю, неврахування області застосування розроблюваного ПЗ, ігнорування етапів життєвого циклу програмного забезпечення і, нарешті, необгрунтоване їх використання в

різнопланових процедурах прийняття виробничих рішень, істотно підірвало до них довіру розробників і користувачів ПЗ. Проте, аналіз технологічного досвіду лідерів виробництва ПО показує, наскільки дорого обходиться недосконаліть ненаукового прогнозу разрешимости і трудовитрат, складності програм, негнучкість контролю та управління їх розробкою та багато іншого, що вказує на відсутність наскрізної методичної підтримки і призводить зрештою до його невідповідності вимогам користувача, необхідному стандарту і до подальшої болючою і трудомісткою його переробці. Ці обставини, вимагають ретельного відбору методик, моделей, методів оцінки якості ПЗ, врахування обмежень їх придатності для різних життєвих циклах і в межах життєвого циклу, встановлення порядку їх спільного використання, застосування надмірної разномодельного дослідження одних і тих же показників для підвищення достовірності поточних оцінок, накопичення та інтеграції різнорідної метричної інформації для прийняття своєчасних виробничих рішень і заключної сертифікації продукції .

Коротко розглянемо метрики складності. Однією з основних цілей науково -технічної підтримки є зменшення складності ПЗ. Саме це дозволяє знизити трудомісткість проектування, розробки, випробувань і супроводу, забезпечити простоту і надійність виробленого ПЗ. Цілеспрямоване зниження складності ПЗ являє собою багатокрокового процедуру і вимагає попереднього дослідження існуючих показників складності, проведення їх класифікації та співвіднесення з типами програм та їх місцем розташування в життєвому циклі.

Теорія складності програм орієнтована на управління якістю ПЗ і контроль її еталонної складності в період експлуатації. В даний час різноманіття показників (у тій чи іншій мірі описують складність програм) настільки велике, що для їх вживання потрібно попереднє упорядкування . У ряді випадків задовольняються трьома категоріями метрик складності. Перша категорія визначається як словникова метрика, заснована на метричних співвідношеннях Холстеда, цикломатическая заходи Мак- Кейба і вимірах Тейер . Друга категорія орієнтована на метрики зв'язків, що відображають складність відносин між компонентами системи - це метрики Уіна і Вінчестера. Третя категорія включає семантичні метрики, пов'язані з архітектурним побудовою програм та їх оформленням .

Відповідно до іншої класифікації, показники складності діляться на дві групи: складність проектування і складність функціонування . Складність проектування, яка визначається розмірами програми, кількістю оброблюваних змінних, трудомісткістю і тривалістю розробки та ін факторами, аналізується на основі трьох базових компонентів : складність структури програми, складність перетворень (алгоритмів), складність даних. У другу групу показників віднесені тимчасова, програмна й інформаційна складності, що характеризують експлуатаційні якості ПЗ.

Існує ще ряд підходів до класифікації заходів складності, проте вони, фіксуючи приватні сторони досліджуваних програм, не дозволяють (нехай з великим допущенням) відобразити загальне, то, чиї виміри можуть лягти в основу виробничих рішень .

Загальним, инвариантно властивим будь-якому ПО (і пов'язаної з його коректністю), є його СТРУКТУРА . Важливо пов'язати це обставина з певним значенням структурної складності в сукупності заходів складності ПЗ. І більше того, при аналізі структурної складності доцільно обмежитися тільки її топологічними заходами, тобто заходами, в основі яких лежать топологічні характеристики граф - моделі програми . Ці заходи задовольняють переважній більшості вимог, що пред'являються до показників : спільність застосовності, адекватність розглянутого властивості, істотність оцінки, спроможність, кількісне вираження, відтворюваність вимірювань, мала трудомісткість обчислень, можливість автоматизації оцінювання .

Саме топологічні міри складності найчастіше застосовуються у фазі досліджень, формує рішення з управління виробництвом (в процесах проектування, розробки і випробувань) і становлять доступний і чутливий еталон готової продукції, контроль якого необхідно регулярно здійснювати в період її експлуатації.

Першою топологічної мірою складності є цикломатическая міра Мак- Кейба . В її основі лежить ідея оцінки складності ПЗ за кількістю базисних шляхів в її керуючому графі, тобто таких шляхів, компонуючи які можна отримати всілякі шляхи з входу графа в виходи . Цикломатичне число l (G) орграфа G з n - вершинами, m - дугами і p - компонентами зв'язності є величина l (G) = m - n + p .

Має місце теорема про те, що число базисних шляхів в орграфе одно його

цикломатическая числу, збільшеному на одиницю . При цьому, цикломатическая складністю ПО P з керуючим графом G називається величина n (G) = l (G) +1 = m - n + 2 . Практично цикломатическая складність ПО дорівнює числу предикатів плюс одиниця, що дозволяє обчислювати її без побудови керуючого графа простим підрахунком предикатів . Дана міра відображає психологічну складність ПЗ.

До достоїнств заходи відносять простоту її обчислення і повторюваність результату, а також наочність і змістовність інтерпретації. Як недоліки можна відзначити: нечутливість до розміру ПО, нечутливість до зміни структури ПО, відсутність кореляції зі структурованістю ПО, відсутність відмінності між конструкціями Розвилка і Цикл, відсутність чутливості до вкладеності циклів . Недоліки цикломатическая заходи призвело до появи її модифікацій, а також принципово інших заходів складності.

Дж. Майерс запропонував як міри складності інтервал [n 1 ¸ n 2], де n 1 - цикломатическая міра, а n 2 - число окремих умов плюс одиниця . При цьому, оператор DO вважається за одну умову, а CASE cn - исходами за n - 1 - умов. Введена міра отримала назву інтервального заходом.

У. Хансену належить ідея брати в якості міри складності ПО пару { цикломатическая число, число операторів } . Відома топологічна міра Z (G), чутлива до структурованості ПЗ. При цьому, вона Z (G) = V (G) (дорівнює цикломатическая складності) для структурованих програм і Z (G) > V (G) для неструктурованих . До варіантів цикломатическая міри складності відносять також міру M (G) = (V (G), C, Q), де C - кількість умов, необхідних для покриття керуючого графа мінімальним числом маршрутів, а Q - ступінь зв'язності структури графа програми та її протяжність .

До заходів складності, враховує вкладеність керуючих конструкцій, відносять тестуючу міру M і міру Харрісона - Мейджела, що враховують рівень вкладеності і протяжності ПО, міру Пивоварського - цикломатическая складність і глибину вкладеності, і міру Мак- Клур - складність схеми розбиття ПО на модулі з урахуванням вкладеності модулів і їх внутрішньої складності .

Функціональна міра складності Харрісона - Мейджела передбачає приписування кожній вершині графа своєї власної складності (первинної) і розбиття графа на сфери впливу предикатних вершин. Складність сфери назива-

ють наведеною і складають її з первинних складнощів вершин, що входять в сферу її впливу, плюс первинну складність самої предикатной вершини . Первинні складності обчислюються всіма можливими способами. Звідси функціональна міра складності ПО є сума наведених складнощів всіх вершин керуючого графа.

Міра Пивоварського ставить метою врахувати в оцінці складності ПО відмінності не тільки між послідовними і вкладеними керуючими конструкціями, а й між структурованими і неструктурованими програмами. Вона виражається відношенням N (G) = n * (G) + S Pi, де n * (G) - модифікована цикломатическая складність, обчислена так само, як і V (G), але з однією відмінністю : оператор CASE з n - виходами розглядається як один логічний оператор, а не як n - 1 операторів. Pi - глибина вкладеності i - тієї предикатной вершини .

Для підрахунку глибини укладення предикатних вершин використовується число сфер впливу. Під глибиною вкладеності розуміється число всіх сфер впливу предикатів, які або повністю утримуватися в сфері розглянутої вершини, або перетинаються з нею. Глибина вкладеності збільшується за рахунок вкладеності не самих предикатів, а сфер впливу. Порівняльний аналіз цикломатическая і функціональних заходів з обговорюваної для десятка різних керуючих графів програми показує, що при нечутливості інших заходів цього класу, міра Пивоварського зростає при переході від послідовних програм до вкладених і далі до неструктурованих .

Міра Мак- Клур призначена для управління складністю структурованих програм у процесі проектування. Вона застосовується до ієрархічним схемами розбивки програм на модулі, що дозволяє вибрати схему розбиття з меншою складністю задовго до написання програми . Метрикою виступає залежність складності програми від числа можливих шляхів виконання, числа керуючих конструкцій і числа змінних (від яких залежить вибір шляху) . Методика розрахунку складності по Мак-Клур чітко орієнтована на добре структуровані програми.

Тестуючої мірою M називається міра складності, яка задовольняє таким умовам

1. Міра складності простого оператора дорівнює 1 ;

2. M ({ F1 ; F2 ; ; Fn }) = Ein M (Fi);

3. M (IF P THEN F1 ELSE F2) = 2 MAX (M (F1), M (F2)) ;

4. M (WHILE P DO F) = 2 M (F).

Міра зростає з глибиною вкладеності і враховує протяжність програми . До тестирующей міру близько примикає міра на основі регулярних вкладень . Ідея цієї міри складності програм полягає в підрахунку сумарного числа символів (операндів, операторів, дужок) в регулярному виразі з мінімально необхідним числом дужок, що описує керуючий граф програми . Всі заходи цієї групи чутливі до вкладеності керуючих конструкцій і до протяжності програми . Однак зростає рівень трудомісткості обчислень.

Розглянемо заходи складності, що враховують характер розгалужень . В основі вузловий заходи Вудворда, Хедлі лежить ідея підрахунку топологічних характеристик потоку управління . При цьому, під вузловий складністю розуміється число вузлів передач управління . Дана міра відстежує складність лінеаризації програми і чутлива до структуризації (складність зменшується). Вона застосовна для порівняння еквівалентних програм, переважніше заходи Холстеда, але по спільності поступається міру Мак- Кейба .

Топологічна міра Чена висловлює складність програми числа перетинів кордонів між областями, утвореними блок - схемою програми . Цей підхід застосовується лише до структурованим програмами, що допускає лише послідовне з'єднання керуючих конструкцій . Для неструктурованих програм міра Чена істотно залежить від умовних і безумовних переходів . У цьому випадку можна вказати верхню і нижню межі міри. Верхня - є m +1, де m - число логічних операторів при їх гніздовий вкладеності . Нижня - дорівнює 2 . Коли керуючий граф програми має тільки одну компоненту зв'язності, міра Чена збігається з цикломатическая мірою Мак- Кейба .

Метрики Джілба оцінюють складність графоорієнтірованних модулів програм відношенням числа переходів за умовою до загального числа виконуваних операторів . Добре зарекомендувала себе метрика, що відноситься число міжмодульних зв'язків до загального числа модулів. Названі метрики використовувалися для оцінки складності еквівалентних схем програм, особливо схем Янова .

Використовуються також міри складності, що враховують історію обчислень, характер взаємодії модулів і комплексні заходи .

Сукупність цикломатическая заходів придатна для оцінювання складності первинних формалізованих специфікацій, які задають в сукупності вихідні дані, цілі та умови побудови шуканого ПЗ. Оцінка цієї первинної програми або порівняння декількох альтернативних її варіантів дозволить спочатку гармонізувати процес розробки ПЗ та від стартової точки контролювати і управляти його поточної результуючої складністю.

Оскільки обробка даних зачіпає наше життя все більшою мірою, помилки ЕОМ можуть тепер мати такі наслідки, як нанесення матеріального збитку, порушення секретності і багато інших, включаючи смерть.

Висока вартість програмного забезпечення - багато в чому наслідок низької надійності. При збільшенні продуктивності програміста (якщо вимірювати її тільки швидкістю розробки та кодування програми) вартість істотно не зменшується. Спроби збільшити продуктивність програміста можуть у деяких випадках навіть підвищити вартість . Найкращий шлях скорочення вартості - у зменшенні вартості його тестування і супроводу. А це може (Під супроводом розуміється будь-яке продовження роботи з програмним продуктом, такі як зміни, доповнення тощо з метою забезпечення його подальшої працездатності та відповідності його вимогам часу) . А це може бути досягнуто не за рахунок інструментів, покликаних збільшити швидкість програмування, а лише в результаті розробки засобів, що підвищують коректність і чіткість при створенні програмного забезпечення. Основні принципи проектування.

2.2 Застосування метрик інтегрованих властивостей у ході дослідження потенційно-небезпечних дефектів цільових програм

В еру стрімкого розвитку IT, ефективність кібернетичного впливу залежить не тільки від можливості знайти вразливість, але й від часових характеристик, адже вразливість, яка сьогодні знайдена, завтра може бути вже усунутою. Для забезпечення кращих часових показників необхідно провести правильний розподіл робочих ресурсів для аналізу та розробки засобів впливу - і в ручну таке завдання зайняло б недопустиму кількість часу. Тому використання метрк коду в цьому випадку може допомогти у виборі тих участків, областей коду, де ймовірність успішного впровадження більша.

Тому з однієї сторони код потрібно аналізувати на предмет наявності потенційно-небезпечних дефектів коду, а з іншої - вираховувати його метрики та спрямовувати зусилля в тому напрямку, в якому для цього сприяють і самі властивості, характеристики вихідного тексту.

Як відомо дефект переповнення буферу не завжди може бути використаним для використання в цілях кібернетичного впливу. Але можна припустити, що в слабо структурованому коді, який містить запутану логіку, велику кількість переходів, розгалужень та змінних набагато легше допустити критичну помилку, яка надасть можливість провести кібернетичний вплив від зацікавленої сторони.

Тому пропоную аналізувати код за допомогою наступних метрик:

- метрики Холстеда (розмірність коду, словника)
- метрики Маккейба (цикломатична складність)
- метрики Джилба (складність розгалуження коду)

Метрика Холстеда

До групи оцінок розміру програм можна віднести також метрику Холстеда . За базу прийнятий підрахунок кількості операторів і операндів використовуються у програмі, тобто визначення розміру програми .

Основу метрики Холстеда складають чотири вимірювані характеристики програми : h1 - число унікальних операторів програми, включаючи символи

- роздільники, імена процедур і знаки операцій (словник операторів) ; h2 - число унікальних операндів програми (словник операндів) ; N1 - загальне число операторів в програмі N2 - загальне число операндів в програмі.

Спираючись на ці характеристики, одержувані безпосередньо при аналізі вихідних текстів програм, Холстед вводить такі оцінки

Словник програми h = h1 + h2

Довжину програми N = N1 + N2

Обсяг програми V = N log2 h

Сенс оцінок h і N досить очевидний, тому докладно розглянемо тільки характеристику V.

Кількість символів, що використовуються при реалізації деякого алгоритму, визначається в числі інших параметрів і словників програми h, що представляє собою мінімально необхідне число символів, що забезпечують реалізацію алгоритму .

Далі Холстед вводить h * - теоретичний словник програми, тобто словниковий запас, необхідний для написання програми з урахуванням того, що необхідна функція вже реалізована в даній мові і, отже, програма зводиться до виклику цієї функції. Наприклад, згідно Холстеду можливе здійснення процедури виділення простого числа могло б виглядати так :

CALL SIMPLE (X, Y),

де Y- масив чисельних значень, що містять шукане число X.

Теоретичний словник в цьому випадку буде складатися з

n1 * : { CALL, SIMPLE (...) } n1 * = 2 ;

n2 * : { X, Y }, h2 * = 2 ;

h * = h1 * + h2 * дорівнюватиме 4 .

Використовуючи h *, Холстед вводить оцінку V * : V * = h * log2h *,

за допомогою якої описується потенційний обсяг програми, соотвествующий максимально компактно реалізує даний алгоритм.

Метрика Маккейба

Друга найбільш представницька група оцінок складності програм - метрики складності потоку керування програм. Як правило, за допомогою цих оцінок оперують або щільністю керуючих переходів усередині програм, або взаємозв'язками цих переходів .

І в тому і в іншому випадку стало традиційним уявлення програм у вигляді керуючого орієнтованого графа G (V, E), де V - вершини, відповідні операторам, а E - дуги, відповідні переходам . У дузі (U, V) - вершина V є вихідною, а U - кінцевої . При цьому U безпосередньо випливає V, а V безпосередньо передує U. Якщо шлях від V до U складається більш ніж з однієї дуги, тоді U слід за V, а V передує U.

Вперше графічне представлення програм було запропоновано МакКейб . Основний метрикою складності він пропонує вважати цикломатическая складність графа програми, або, як ще називають, цикломатичне число МакКейб, що характеризує трудомісткість тестування програми .

Для обчислення цикломатическая числа МакКейб Z (G) застосовується формула

Z (G) = l - v +2 p,

де l - число дуг орієнтованого графа G ; v - число вершин ; p -число компонентів зв'язності графа.

Число компонентів зв'язності графа можна розглядати як кількість дуг, які необхідно додати для перетворення графа сільносвязний . Сільносвязний називається граф, будь-які дві вершини якого взаємно досяжні. Для графів коректних програм, тобто графів, що не мають недосяжних від точок входу дільниць і « висячих » входу і виходу, сільносвязний граф, як правило, виходить шляхом замикання однієї вершини, що позначає кінець програми на вершину, що позначає точку входу в цю програму.

По суті Z (G) визначає число лінійно незалежних контурів у сільносвязний графі . Інакше кажучи, цикломатичне число МакКейб показує необхідне число проходів для покриття всіх контурів Сільносвязанная графа або кількість тестових прогонів програми, необхідних для вичерпного тестування за критерієм « працює кожна гілка» .

Для програми цикломатичне число при l = 10, v = 8, n = 1 визначиться як Z (G) = 10-8 +2 = 4 .

Таким чином, є Сільносвязанная граф з чотирма лінійно незалежними контурами :

a - b - c - g - e - h - a ;

a - b - c - e - h - a ;

a - b - d - f - e - h - a ;

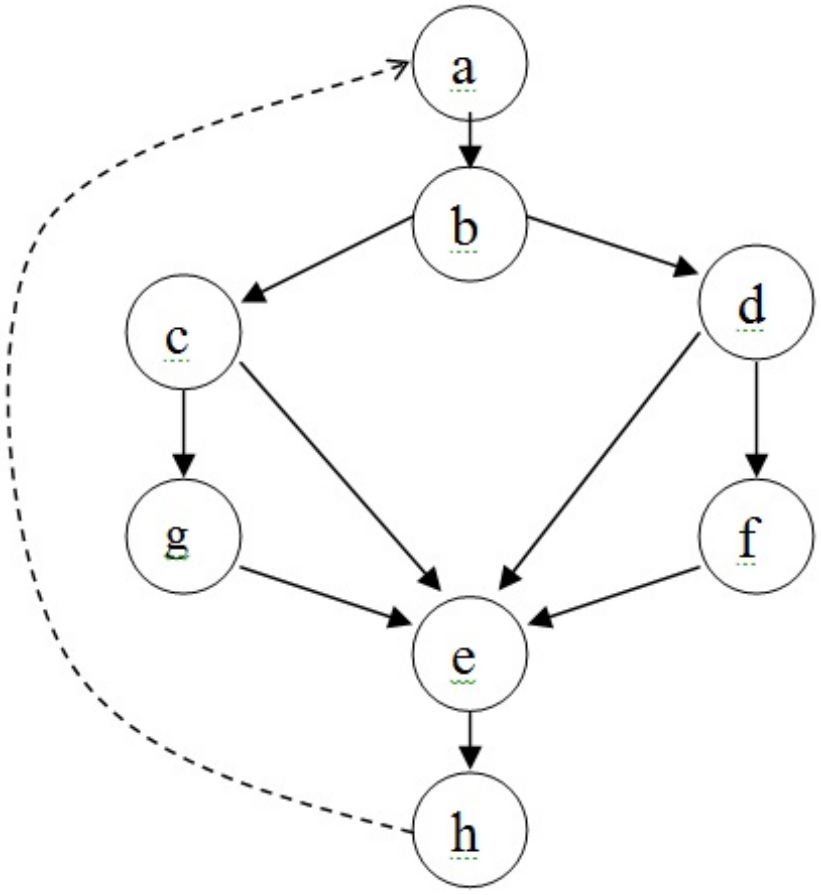

Рис. 2.1: граф в метриці Маккейба

a - b - d - e - h - a ;

Розглянемо метрику складності програми, що отримала назву « підрахунок точок перетину «, авторами якої є М Вудвард, М Хенель і Д Хидли . Метрика орієнтована на аналіз програм, при створенні яких використовувалося неструктурні кодування на таких мовах, як мова асемблера і фортран . Вводячи цю метрику, її автори прагнули оцінити взаємозв'язку між фізичними місцеположеннями керуючих переходів .

Структурний кодування припускає використання обмеженого безлічі керуючих структур в якості первинних елементів будь-якої програми . У класичному структурному кодуванні, що базується на роботах професора Ейндховенського технологічного університету (Нідерланди) Е. Дейкстри, оперують тільки трьома такими структурами : проходженням операторів, розвилкою з операторів, циклом над оператором. всі ці різновиди зображуються найпростішими планарними графами програм. За правилами структурного кодування будь-яка програма складається шляхом вибудовування ланцюжків з 3х

згаданих структур або приміщення однієї структури в іншу. Ці операції не порушують планарности графа всієї програми .

У графі програми, де кожному оператору відповідає вершина, тобто не виключені лінійні ділянки, при передачі управління від вершини a до b номер оператора a дорівнює min (a, b), а номер оператора b - max (a, b) . Тоді перетин дуг з'являться, якщо

min (a, b) < min (p, q) < max (a, b) & max (p, q) > max (a, b) |

0em [] Min (a, b) < max (p, q) < max (a, b) & min (p, q) < min (a, b) .

Іншими словами, точка перетину дуг виникає у разі виходу управління за межі пари вершин (a, b).

Кількість точок перетину дуг графа програми дає характеристику неструктурованості програми.

Метрика Джилба

Однією з найбільш простих, але досить ефективних оцінок складності програм є метрика Т. Джилба, в якій логічна складність програми визначається як насиченість програми виразами IF_THEN_ELSE . При цьому вводяться дві характеристики :

CL - абсолютна складність програми, що характеризується кількістю операторів умови; cl - відносна складність програми, що характеризується насиченістю програми операторами умови, тобто cl визначається як відношення CL до загального числа операторів. Використовуючи метрику Джилба, її доповнили ще однією складовою, а саме характеристикою максимального рівня вкладеності оператора CLI, що дозволило застосувати метрику Джилба до аналізу циклічних конструкцій.

Великий інтерес представляє оцінка складності програм за методом граничних значень .

Введемо кілька додаткових понять, пов'язаних з графом програми .

Нехай G = (V, E) - орієнтований граф програми з єдиною початковою і єдиною окнечной вершинами. У цьому графі число входять до вершину дуг називається негативною ступенем вершини, а число що виходять з вершини дуг - позитивної ступенем вершини . Тоді набір вершин графа можна розбити на дві групи:

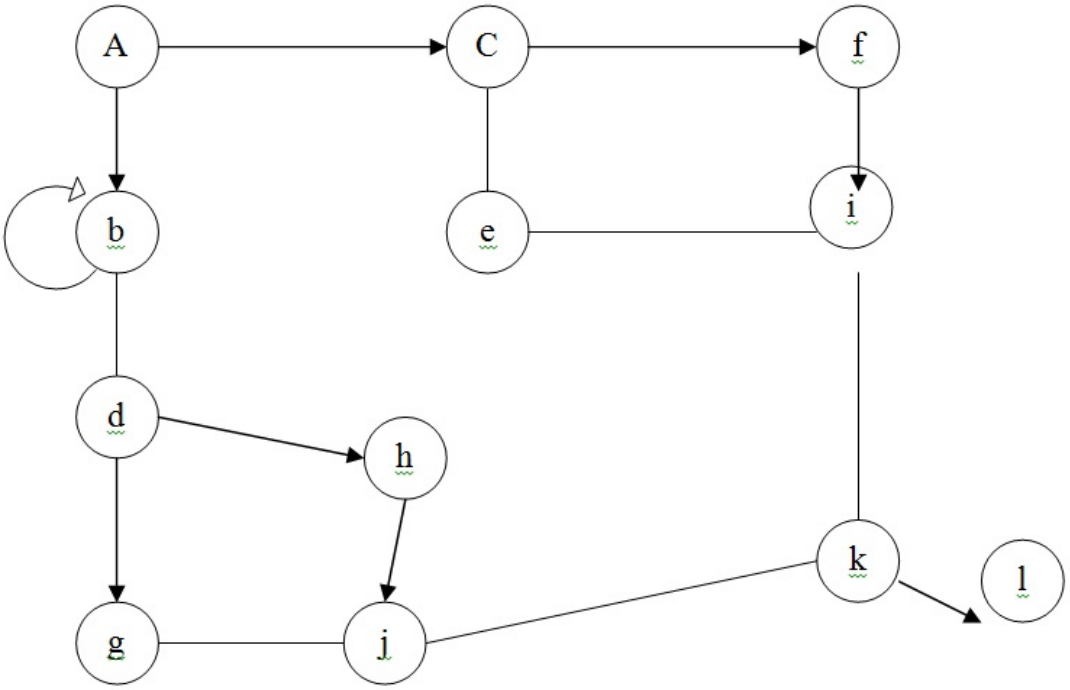

Рис. 2.2: граф в метриці Джилба

вершини у яких позитивна ступінь <= 1 ; вершини у яких позитивна ступінь > = 2 . Вершини першої групи назвемо приймаючими вершинами, а вершини другої групи - вершинами відбору.

Для отримання оцінки за методом граничних значень необхідно розбити

граф G на максимальне число подграфов G ‘, що задовольняють таким умовам:

вхід в подграф здійснюється тільки через вершину відбору; кожен подграф включає вершину (звану нижньою межею подграфа), в яку можна потрапити з будь-якої іншої вершини подграфа . Наприклад, вершина відбору поєднана сама з собою дугою петлею, утворює подграф .

Число вершин, що утворюють такий подграф, одно скоригованої складності вершини відбору.

Кожна приймаюча вершина має скориговану складність, рівну 1, крім кінцевої вершини, скоригована складність якої дорівнює 0 . Скориговані складності всіх вершин графа G підсумовуються, утворюючи абсолютну граничну складність програми . Після цього визначається відносна гранична складність програми :

S0 = 1 - (v - 1) / Sa,

де S0 - відносна гранична складність програми ; Sa - абсолютна гранична складність програми, v - загальне число вершин графа програми .

Таким чином, відносна складність програми дорівнює

S0 = 1 - (11 / 25) = 0,56 .

Інша група метрик складності програм - метрика складності потоку даних, тобто використання, конфігурації і розміщення даних в програмах.

Пара « модуль - глобальна змінна» позначається як (p, r), де p - модуль, що має доступ до глобальної змінної r . Залежно від наявності в програмі реального обігу до змінної r формуються два типи пар « модуль - глобальна змінна» : фактичні і можливі . Можливе звернення до r за допомогою p показує, що область існування r включає в себе p .

Характеристика Aup говорить про те, скільки разів модулі Up дійсно отримали доступ до глобальних змінних, а число Pup - скільки разів вони могли б отримати доступ.

Відношення числа фактичних звернень до можливих визначається

Rup = Aup / Pup

Ця формула показує наближену ймовірність посилання довільного модуля на довільну глобальну змінну. Очевидно, чим вище ця вірогідність, тим вище ймовірність « несанкціонованого » зміни якої-небудь змінної, що може істотно ускладнити роботи, пов'язані з модифікацією програми .

Покажемо розрахунок метрики « модуль - глобальна змінна» . Нехай у програмі є три глобальні змінні і три підпрограми. Якщо припустити, що кожна підпрограма має доступ до кожної з змінних, то ми отримаємо дев'ять можливих пар, тобто Pup = 9 . Далі нехай першим підпрограма звертається до однієї змінної, другий - двом, а третя не звертається ні до однієї змінної. Тоді Aup = 3, Rup = 3 /9.

Алгоритм пошуку залежностей потенційно-небезпечних дефектів програм на основі екстраполяції метричних характеристик вихідних текстів програм для побудови дерева атак

Розглянемо програмний продукт в якості множини вразливостей:

$$Vuln = Vuln_1, Vuln_2, ..., Vuln_N \tag{2.2}$$

Маючи вищенаведені характеристики, та множину потенцыйних вразливостей пропоную створити наступну модель вибору потенційно небезпечних дефектів реакції програм для кібернетичного впливу:

$$P_{Vuln_i} = \frac{V_i Z(G)_i}{Z(G)\frac{V}{C_v}} Rup_i, \tag{2.3}$$

де

- V - Обсяг програми
- Vi - обсяг підпрограми
- Cv - кікькість потенційних вразливостей
- Z(G)i - цикломатична складність підпрограми, в якій знаходиться дефект
- Z(G) - цикломатична складність всього коду досліджуваного проекту
- Rup - кількість звернень потенційно-вразливої ділянки коду до глобальних змінних

Розглянемо кожну властивість поданих метрик в контексті відображення наявності можливості використання дефекту переповнення буферу:

- Обсяг програми - обсяг програми напряму впливає на кількість помилок в ній
- Аналогічним чиним від обсягу підпрограми, в якій знаходиться потенційна вразливість залежить можливість її використання
- Від цикломатичної складності залежить наскільки просто буде проаналізувати логіку коду і швидко розібратись, як саме можна використати дефект
- А від кількості звернень до потенційно-вразливої ділянки коду можна зробити висновок, як локальні дані між собою звязані і чи можливо здійснити вплив на певні управляючі дані через суміжні

Дана ймовірність буде наближеною та неточною, але якщо набір таких ймовірностей збільшувати і розглянути їх як протабульовану функцію $f(V, V_i, C_v, Z(G)_i, Z(G), Rup) = P_{vuln_i}$ то можна спробувати робити прогноз наявності вразливих ділянок при аналізі нового вихідного тексту на основі його метрик коду - що дозволить побудувати ефективне дерево атак.

Дерева атак

Дерева атак - це діаграми, що демонструють, як може бути атакована ціль. Дерева атак використовуються в безлічі областей. В області інформаційних технологій вони застосовуються, щоб описати потенційні загрози у комп'ютерній системі і можливі способи атаки, які реалізують ці загрози. Однак, їх використання не обмежується аналізом звичайних інформаційних систем. Вони також широко використовуються в авіації і обороні для аналізу ймовірних загроз, пов'язаних зі стійкими до спотворень електронними системами.

Збільшується застосування дерев атак в комп'ютерних системах контролю (особливо пов'язаних з енергетичними мережами). Також дерева атаки використовуються для розуміння загроз, пов'язаних з фізичними системами.

Деякі з найбільш ранніх описів дерев атак знайдені в доповідях і статтях Брюса Шнайера, технічного директора Counterpane Internet Security. Шнайер був безпосередньо залучений в розробку концептуальної моделі дерев атаки і зіграв важливу роль в її поширенні. Тим не менш, в деяких ранніх опублікованих статтях по деревах атак висловлюються припущення про залученість Агентства Національної Безпеки в початковий етап розробки.

Дерева атак дуже схожі на дерева загроз. Дерева загроз були розглянуті в 1994 роки Едвардом Аморосо.

Дерева атак це мультирівневих діаграми, що складаються з одного кореня, листя і нащадків. Будемо розглядати вузли знизу вгору. Дочірні вузли це умови, які повинні виконуватися, щоб батьківський вузол також перейшов в справжній стан. Коли корінь переходить у справжній стан, атака успішно завершена. Кожен вузол може бути приведений у справжній стан тільки його прямими нащадками.

Вузол може бути дочірнім для іншого вузла, в цьому випадку, логічно, що для успіху атаки потрібно кілька кроків. Наприклад, уявіть клас з комп'ютерами, де комп'ютери прикріплені до парт. Щоб вкрасти один з них необхідно

або перерізати кріплення, або відкрити замок. Замок можна відкрити відмичкою або ключем. Ключ можна отримати шляхом погроз його власнику, через підкуп власника або ж просто вкрасти його. Таким чином, можна намалювати чотирирівневої дерево атаки, де одним із шляхів буде: Підкуп власника ключа-Отримання ключа-Відмикання замку-Винос комп'ютера.

Також слід враховувати, що атака, описана у вузлі може зажадати, щоб одна або декілька з безлічі атак, описаних в дочірніх вузлах були успішно проведені. Вище ми показали дерево атаки тільки зі зв'язком типу АБО між нащадками вузла, але умова І також може бути введено, наприклад в класі є електронна сигналізація, яка повинна бути відключена, але тільки в тому випадку, якщо ми вирішимо перерізати кріплення. Замість того, щоб робити відключення сигналізації дочірнім вузлом для перерізання кріплення, обидва завдання можна просто логічно підсумувати, створивши шлях (Відключення сигналізації І Перерізання кріплення)-Винос Комп'ютера.

Дерева атак також пов'язані із створенням дерева помилок. Метод побудови дерева помилок використовує булеві вирази для створення умов, при яких дочірні вузли забезпечують виконання батьківських вузлів.

Включає апріорні ймовірності в кожен вузол, можливо зробити підрахунок ймовірностей для вузлів, що знаходяться вище по правилу Байеса . Однак, в реальності, точні оцінки ймовірності або недоступні, або занадто дорогі для обчислення. У разі динамічної комп'ютерної безпеки (тобто з урахуванням атакуючих) випадкові події не є незалежними, отже, прямий Байєсівський аналіз не підходить.

Так як Байєсовські аналітичні техніки, що використовуються в аналізі дерев помилок не можуть бути правильно застосовані до дерев атак, аналітики використовують інші техніки для визначення яким шляхом піде даний атакуючий. Ці техніки включають порівняння можливостей атакуючого (час, гроші, навички, обладнання) з ресурсами, що вимагаються для даної атаки. Атаки, які вимагають повної віддачі від атакуючого або навіть знаходяться за межами його можливостей куди менш імовірні, ніж дешеві і прості атаки. Ступінь, в якій атака задовольняє цілям атакуючого також впливає на його вибір. З двох можливих атак зазвичай вибирається та, що більшою мірою задовольняє цілям атакуючого.

Дерева атак можуть стати вкрай складними, особливо при розгляді кон-

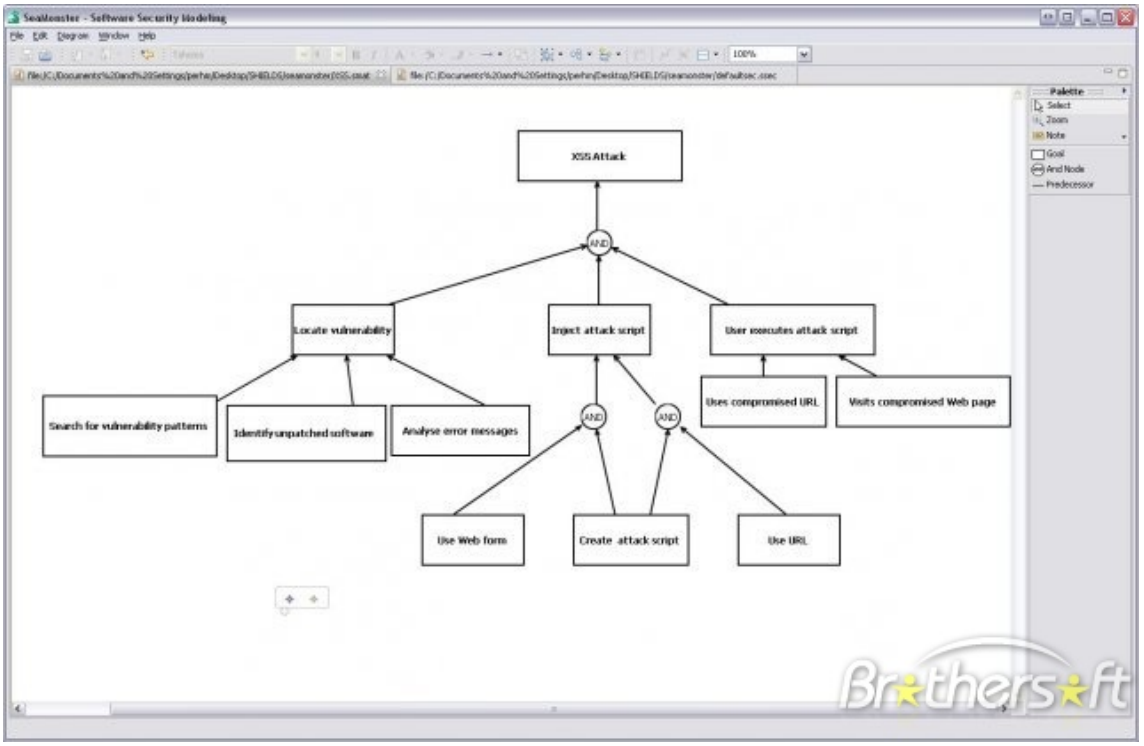

Рис. 2.3: приклад дерева атак

кретних атак. Повне дерево атаки може містити сотні або тисячі різних шляхів, всі з яких призводять до успіху атаки. Але навіть при такому розкладі, ці дерева вкрай корисні для визначення існуючих загроз і методів їх запобігання.

Дерева атак можуть бути використані для визначення стратегії забезпечення інформаційної безпеки. Також, слід враховувати, що реалізація цієї стратегії сама по собі вносить зміни в дерево атаки. Наприклад, захистом від комп'ютерних вірусів може служити заборона системного адміністратора безпосередньо змінювати існуючі файли і папки, замість цього вимагаючи використання файлового менеджера. Це додає в дерево атаки використання недоліків чи експлойтів файлового менеджера.

Висновки

Отже, проаналізувавши обчислення метрик інтегрованих властивостей вихідних кодів програмних засобів дозволяють зекономити час та інші ресурси в плані організації та розробки засобів кібернетичного впливу та побудови дерев атак тому підготовка фахівців в даній області є необхідною, як і розширення навчальної бази для досягнення даної цілі – на даний момент напрацьована невелика науково-методична база в даній області.

3. ПРОГРАМНИЙ МОДУЛЬ ВИЗНАЧЕННЯ МЕТРИК ЦІЛЬОВОЇ ПРОГРАМИ ДЛЯ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ДОСЛІДЖЕННЯ ЇЇ ПОТЕНЦІЙНО-НЕБЕЗПЕЧНИХ ДЕФЕКТІВ

3.1 Структурна схема алгоритму та основні функціональні елементи

Підчас аналізу предметної області мною була запропонована структурна схема програмно-технічного комплексу виявлення залежностей між метриками інтегрованих властивостей програм. (Рис 3.1)

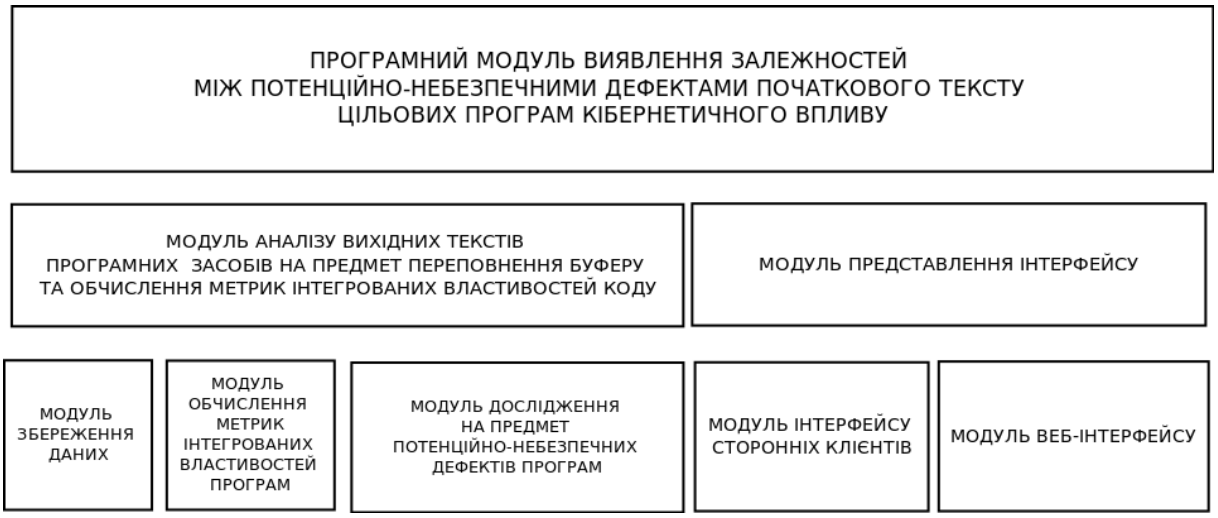

Рис. 3.1: Структурна схема модулю визначення метрик цільової програми

Модулі програмно-технічного комплексу:

1. Модуль аналізу вихідних текстів програмних засобів на предмет переповнення буферу та обчислення метрик інтегрованих властивостей коду;

2. Модуль представлення інтерфейсу.

Модуль аналізу вихідних текстів програмних засобів включає в себе декілька підмодулів:

- Модуль обчислення метрик інтегрованих властивостей програм;

- Модуль досліджнення на предмет потенційно-небезпечних дефектів програм;

- Модуль збереження даних.

В свою чергу, модуль представлення інтерфейсу містить 2 підмодулі:

- Модуль веб-інтерфейсу

- Модуль інтерфейсу сторонніх клієнтів

В цілому програмний модуль реалізовано на базі *Google App Engine* - платформі, що являє собою хмарний сервіс.

Зараз багато говорять про хмарні сервіси та хмарні обчислення. Економічну ефективність хмарних сервісів вже оцінив бізнес, а їх зручність – пересічний користувач. Часом можна зустріти компанію або користувачів, які активно працюють з хмарними сервісами, не підозрюючи про те, що вони знаходяться "на передньому краї"сучасних *IT*-технологій, а їх дії відповідають найновішим трендам *IT*-галузі.

Під хмарними обчисленнями розуміють інструменти, які доступні користувачу через інтернет або локальну мережу у вигляді веб-сервісу. Управляти такими інструментами можна лише за допомогою браузера. Таким чином можна отримати віддалений доступ до певних ресурсів (обчислювальних ресурсів, програм, даних) – хмарних сервісів. Основна перевага хмарних обчислень полягає в тому, що користувач чи клієнт сплачує лише за фактичне використання обчислювальних сервісів, а не за володіння ними. Наприклад, для компанії, яка лише починає працювати на ринку і для якої не потрібні потужні ресурси, але яка потенційно планує активно розширюватись, це особливо зручно. Купити готові програмні рішення та відповідне апаратне забезпечення (комп'ютери, сервера) буде доволі дорого на початку, а ось оплата лише за використані ресурси є доступною практично для всіх. Водночас при розширенні клієнтської бази та збільшенні необхідності щодо обчислювальних ресурсів компанії буде достатньо лише заявити про це своє бажання своєму постачальнику хмарних технологій та перейти на інший тарифний план. В результаті цього з мінімальними затратами часу і сил клієнт отримає ресурси, які відповідають його запитам у кожний конкретний момент часу.

Однією з перших компаній, яка запропонувала своїм клієнтам хмарний сервіс була компанія *Salesforce.com* – клієнтам *Salesforce.com* було запропоновано не купувати у компанії готові рішення (які були досить дорогими), а підписуватись на послуги *Salesforce.com*, отримуючи цілий ряд переваг. В результаті такого аутсорсингу інформаційних послуг, клієнти *Salesforce.com* отримували можливість використання продуктів *Salesforce.com*, встановлених на їх серверах, захист від збоїв, технічну підтримку.

Salesforce.com почала пропонувати свої рішення в 2000 році, а уже в 2005 році компанія *Amazon* вийшла на цей ринок із рішенням *"Amazon Web Services"*. Починаючи з 2006 року одним з найбільших представників ринку хмарних сервісів стала компанія Google з її продуктом *"Google Apps"*, а згодом і *Microsoft* з *"Azure Services Platform"*.

Хмарні обчислення найчастіше класифікують відповідно до їх змісту і призначення. Хмарні технології можна розділити на такі види

1. Програмне забезпечення як сервіс *("Software as a Service"або "SaaS")* – програмне забезпечення у вигляді сервісу, доступ до якого здійснюється через веб. Саме до цього виду можна віднести більшість хмарних сервісів для користувачів;

2. Інфрастуктура як сервіс *("Infrastructure as a Service"або "IaaS")* – надання платформи віртуалізації у форматі послуги;

3. Платформа як сервіс *("Platform as a Service"або "PaaS")* – надання платформи для розробки, тестування, розсортування і підтримки вебдодатків

Види хмарних технологій *IaaS* та *PaaS* найчастіше використовуються розробниками для створення нових продуктів та *IT*-фахівцями, які працюють над вирішенням бізнес-задач.

Одним з перших і найпопулярніших хмарних інструментів є продукт *Amazon Web Services*, до складу якого увійшли засоби для збереження *(Simple Storage Service, SimpleDB)* и обробки *(EC2, EC2 MapReduce)*, а також послуги для організації хмарних мереж і хмарних додатків, наприклад, *Amazon Mechanical Turk (Mturk), Amazon Historical Pricing, Amazon Flexible Payments Service (FPS)* .

Компанія *Google* пропонує хмарне середовище *Google App Engine*, в якому можна запускати додатки, написані на мові *Python* із додатковою підтримкою фреймворків *Django, WebOb и PyYAML*, а також додатки на *Java*.

Ще одне рішення від *Google* – набір хмарних сервісів *Google Apps*, до складу якого входять електронна пошта *Gmail*, веб-календар *Google Calendar*, інструмент комунікації *Talk*, система роботи з документами *Google Docs* і сервіс для створення сайтів *Sites*. Всі ці сервіси інтегровані між собою і підтримують взаємодію за допомогою скриптів на *javascript*, які підтримуються для всіх клієнтів *Apps Premier* и *Education Edition*.

Хмарні сервіси від*Microsoft* включають хмарну платформу *Windows Azure Platform* и набір хмарних сервісів *(.NET, SQL Azure, Live, SharePoint, Dynamics CRM). Windows Azure* представляє собою платформу і інфраструктуру для запуску *Windows*-додатків в хмарному середовищі.

Не дивлячись на цілий ряд переваг хмарних сервісів (доступність, економічність і ефективність, гнучкість) існують й проблеми для їх розповсюдження і використання. Вочевидь, хмарні сервіси вимагають постійного інтернет-з'єднання, що є не завжди доступним, особливо у регіонах. Потенційні клієнти часто не довіряють хмарним сервісам через можливі проблеми з безпекою даних. Окрім того, розповсюдження хмарних сервісів вимагає наявності у країні потужних центрів обробки даних. І головне – не всі інструменти або програми можуть бути доступними у віддаленому режимі.

Модуль аналізу вихідних текстів програмних засобів

Даний модуль реалізує аналіз метричних характеристик досліджуваних програм та потенційно-небезпечних дефектів реакції програм, та на основі цих даних робить кількісну оцінку вразливих участків коду. Дані характеристики зберігаються в *Google Cloud SQL* за допомогою *NDB API*.

Google Cloud SQL-це служба, яка дозволяє створювати, настроювати і використовувати реляційні бази даних, які розміщені в *Google Cloud*. Це повністю керований сервіс, що веде, керує і розпоряджається вашими базами даних, дозволяючи вам зосередитися на ваших додатків і послуг.

Інтерфейс *NDB* зберігає дані об'єкти, відомі як сутності. *NDB* можете згрупувати кілька операцій в одну операцію. Транзакції не можуть досягти успіху, якщо хоча б одна операція в транзакції не завершується успішно

або якщо який-небудь збій операції. Це особливо корисно для розподілених і веб-додатків, де декілька користувачів можуть мати доступ до або маніпулювання даними одночасно.

NDB використовує *Memcache* в якості кеша служби для "гарячих точок"у даних. Якщо додаток зчитує деякі дані часто, *NDB* можете читати їх швидко з кешу.

Як було вище сказано (підрозділ 2.2.4) метрики вихідного тексту в сукупності зі знайденими вразливостями підставляються в формулу (2.3), і обчислюється потенціал вихідного коду, який буде використаний для обчислення ймовірності використання вразливостей даного модулю. В БД зберігаються всі потенціали вихідних текстів, які містять потенційні вразливості разом з ймовірностями, які вираховуються за допомогою лінійної екстраполяції елементів, розміщених в БД. Ці ймовірності являються експериментальними і в процесі дослідження і роботи спеціалістів над певними вихідними текстами проекту можуть бути змінені вручну, для збільшення точності оцінювання. Екстраполювання виконується за допомогою бібліотеки *NumPy*.

NumPy - бібліотека мови *Python* для роботи з гомогенними багатовимірними масивами даних, які індексуються додатніми цілими числами. Гомогенність даних дозволяє значно оптимізувати роботу в порівнянні з стандартними списками мови. Основний тип даних відповідно - array. На базі *NumPy* написано майже усе науково-технічне програмне забеспечення мовою *Python*, зокрема українське ПЗ *OpenOpt* (чисельна оптимізація, автоматичне диференціювання, розв'язування систем рівнянь) *SciPy* (інтеграція, інтерполяція, статистика і т.і.) науково-інженерні *Python*-дистрибутиви *PythonXY, SAGE* (вільні аналоги до *MATLAB, Maple, MatCad, Mathematica* і т.і.)

Оскільки *Python* - інтерпретована мова, математичні алгоритми, часто працюють в ньому набагато повільніше ніж у компільованих мовах, таких як *C* або навіть *Java*. *NumPy* намагається вирішити цю проблему для великої кількості обчислювальних алгоритмів забезпечуючи підтримку багатовимірних масивів і безліч функцій і операторів для роботи з ними. Таким чином будь-який алгоритм який може бути виражений в основному як послідовність операцій над масивами і матрицями працює також швидко як еквівалентний код написаний на *C*.

NumPy можна розглядати як вільну альтернативу *MATLAB*, оскільки мо-

ва програмування *MATLAB* зовні нагадує *NumPy*: обидві вони інтерпретовані, і обидві дозволяють користувачам писати швидкі програми поки більшість операцій проводяться над масивами або матрицями, а не над скалярами. Перевага *MATLAB* у великій кількості доступних додаткових тулбоксів, включаючи такі як пакет *Simulink*. Основні пакети, що доповнюють *NumPy*, це: *SciPy* - бібліотека, що додає більше *MATLAB*-подібної функціональності; *Matplotlib* - пакет для створення графіки в стилі *MATLAB*. Внутрішньо як *MATLAB*, так і *NumPy* базується на бібліотеці *LAPACK*, призначеної для вирішення основних задач лінійної алгебри.

Модуль обчислення метрик інтегрованих властивостей програм

Даний модуль реалізує аналіз вихідного коду програмного забезпечення - а саме обчислення метрик інтегрованих властивостей програмного коду - метрики Холстеда, Маккейба, Джилба, за допомогою побудови абстрактного синтаксичного дерева програмного коду з наступним його аналізом.

Абстрактні синтаксичні дерева зазвичай використовуються при компіляції, коли для нас важлива лише функціональна складова даних, маючи яку ми зможемо згенерувати байткод програми, який буде виконуваті дії, записані у вихідному коді. Основне завдання даної моделі — збереження інформації у такій структурі, яка дозволить легко аналізувати вихідний код, а також генерувати його. Це передбачає збереження якомога більшої кількості інформації у моделі.

Модуль досліджнення на предмет потенційно-небезпечних дефектів програм

Даний модуль реалізує аналіз вихідного коду програмного забезпечення з використанням бібліотеки *Flawfinder*. Ця бібліотека реалізує статичний сканер вихідних текстів програм, написаних на *C/C++*. Виконує пошук функцій, які найчастіше використовуються некоректно, присвоює їм коефіцієнти ризику (спираючись на таку інформацію, як передані параметри) і складає список потенційно вразливих місць, впорядковуючи їх за ступенем ризику.

Модуль збереження даних

Даний модуль містить опис моделей представлення та збереження даних, отриманих в процесі аналізу вихідних текстів програм. Головними сутностями данної моделі є проект, вихідний текст, метрика та вразливість.

```
class Metrix(db.Model):
    mackkeib = db.StringProperty()
    holsted = db.StringProperty()
    jilb = db.StringProperty()
    sloc = db.StringProperty()


class Vulnerability(db.Model):
    vulnerability = db.StringProperty()


class Project(db.Model):
    short = db.StringProperty()
    name = db.StringProperty()
    metrix = db.ReferenceProperty(Metrix, default=None)
    vulnerability = db.ReferenceProperty(Vulnerability, default=None)
    potential = db.FloatProperty()
    p = db.FloatProperty()


class SourceFile(db.Model):
    short = db.StringProperty()
    project = db.ReferenceProperty(Project)
    name = db.StringProperty()
    source = db.BlobProperty()
    metrix = db.ReferenceProperty(Metrix, default=None)
    vulnerability = db.ReferenceProperty(Vulnerability, default=None)
    potential = db.FloatProperty()
    p = db.FloatProperty()


__author__ = 'andrew.vasyltsiv'
```

Модуль представлення інтерфейсу

Модуль представлення інтерфейсу являє собою реалізацію Веб-інтерфейсу та інтерфейсу сторонніх клієнтів. Він реалізований на базі веб-фреймворків *webapp2* та *endpoints*.

Модуль веб-інтерфейсу

Як було сказано вище, модуль веб-інтерфейсу реалізований з використанням фреймворку *webapp2*. Основними його функціями являються:

1. Завантаження вихідних текстів проекту для наступного аналізу
2. Проведення аналізу проекту за допомогою вищенаведених модулів
3. Представлення даних аналізу в графічному та табличному вигляді
4. Корекція результатів аналізу

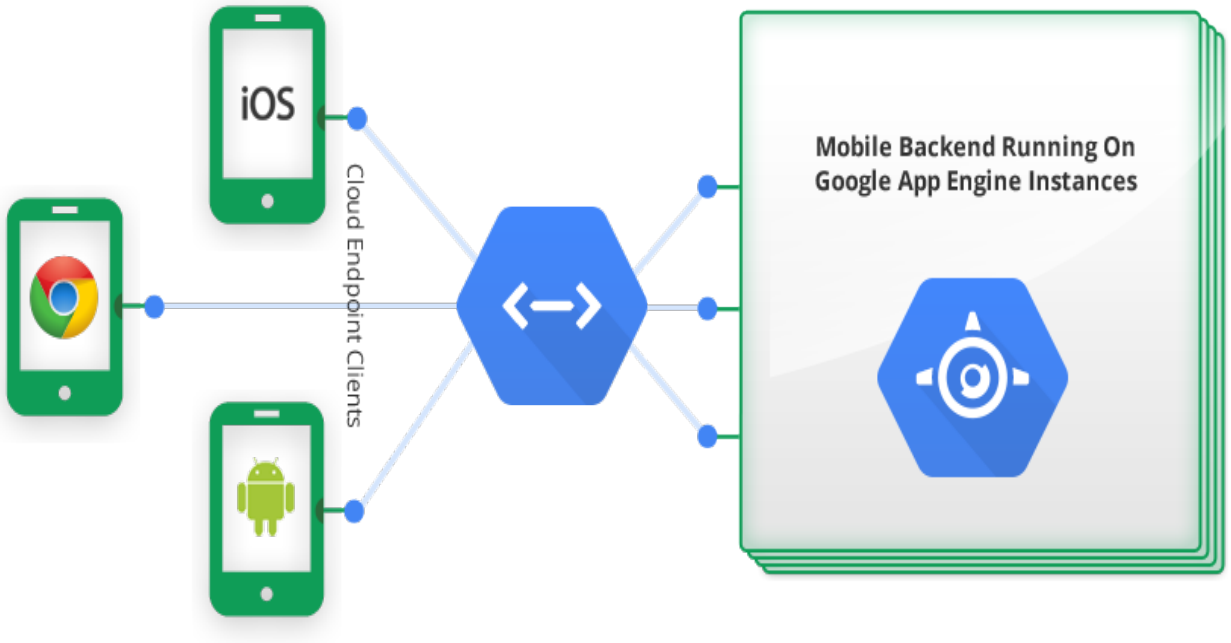

Рис. 3.2: приклад використання *endpoints*

Модуль інтерфейсу сторонніх клієнтів

Модуль інтерфейсу сторонніх клієнтів реалізовано з використанням бібліотеки *endpoints*, являє собою засіб забезпечення взаємодії між інтерфейсним модулем та сторонніми розробками, що дозволяє злегкістю розробляти програмні додатки на різних мовах програмування.

Google Cloud Endpoints складається з інструментів, бібліотек і можливостей, які дозволяють генерувати інтерфейси і клієнтські бібліотеки з *App Engine*, їх також називають *API backend*, для спрощення доступу клієнтів до даних з інших додатків. *Endpoints* полегшують створення веб-серверної частини для веб-клієнтів і мобільних клієнтів, таких як *Android* або *Apple iOS* (Рис. 3.2).

3.2 Інтерфейс користувача

Інтерфейс користувача являє собою Веб-додаток, що дозволяє завантажувати вихідні тексти досліджуваних проектів, та представляти їхні метрики та потенційні вразливості за допомогою діаграм *Google Charts*.

Google Charts забезпечує ідеальний спосіб візуалізації даних на веб-сайті. Бібліотека надає велику кількість готових до використання типів діаграм.

Найбільш поширений спосіб використання *Google Charts* з допомогою скриптів *JavaScript*, які можна вбудовувати в web-сторінки. Спочатку відбувається завантаження *Google Charts* бібліотек, перелік даних для побудови діаграми, проводиться вибір параметрів для настройки діаграми, і, нарешті, створення об'єктів *chart* з *id*, який ви вибираєте. Потім, пізніше у веб-сторінки, можна створити тег * * з допомогою цього ідентифікатора для відображення *Google Chart*.

Це все, що потрібно для початку роботи.

Графіки представлені у вигляді класів *JavaScript*, і *Google Charts* дає багато типів діаграм, які можна використовувати. Зовнішній вигляд за замовчуванням, зазвичай реалізує все, що потрібно в більшості випадків, і завжди існує можливість налаштування діаграми, щоб підігнати зовнішній вигляд до веб-сайту. *Google Charts* є інтерактивними та дозволяють задавати реакцію на події від користувача, що дозволяє підключати їх до створення складних інформаційних панелей, інтегрованих з вашої веб-сторінці. Діаграми будуються з використанням *HTML5/SVG* технології для забезпечення сумісності з різними браузерами (у тому числі *VML* для більш старих версій *IE*) і кросплатформену переносимість *iPad і iPhone і Android*.

Всі типи діаграм, які заповнюються даними, використовуючи клас *DataTable*, що дозволяє легко перемикатися між типами діаграм, щоб дозволить легко підібрати найбільш підходящий тип діаграми для представлення данних. *DataTable* надає методи для сортування, редагування та фільтрації даних, і можуть бути заповнені безпосередньо з веб-сторінки, бази даних, або будь-яких джерел даних, що підтримують інструменти *Charts Datasource* протоколу. (Цей протокол включає в себе *SQL*-подібний мову запитів і реалізується за допомогою електронних таблиць, *Google* зведених таблиць, а також сторонні постачальники даних, таких як *SalesForce*.

Рис. 3.3: приклад використання *Google Charts*

Існує також можливість реалізації власного протоколу для створеннння свого провайдеру даних для своїх додатків (Рис. 3.3).

Окрім бібліотеки *Google Charts*, в модулі інтерфейсу користувача використано бібліотеку *CodeFlower*.

CodeFlower, реалізована на базі *JavaScript*-фреймворку *d3.js*, вона показує репозиторії вихідних кодів з використанням інтерактивного дерева. Кожний вузол являє собою файл, з радіусом пропорційно значенні певної характеристики заданого вузла. Вся візуалізація виконується на стороні клієнта, в *JavaScript*. При наведенні курсору на будь-який вузол відображається назва файлу та його властивості (Рис. 3.4).

Також для представлення вихідних текстів програм для загального огляду

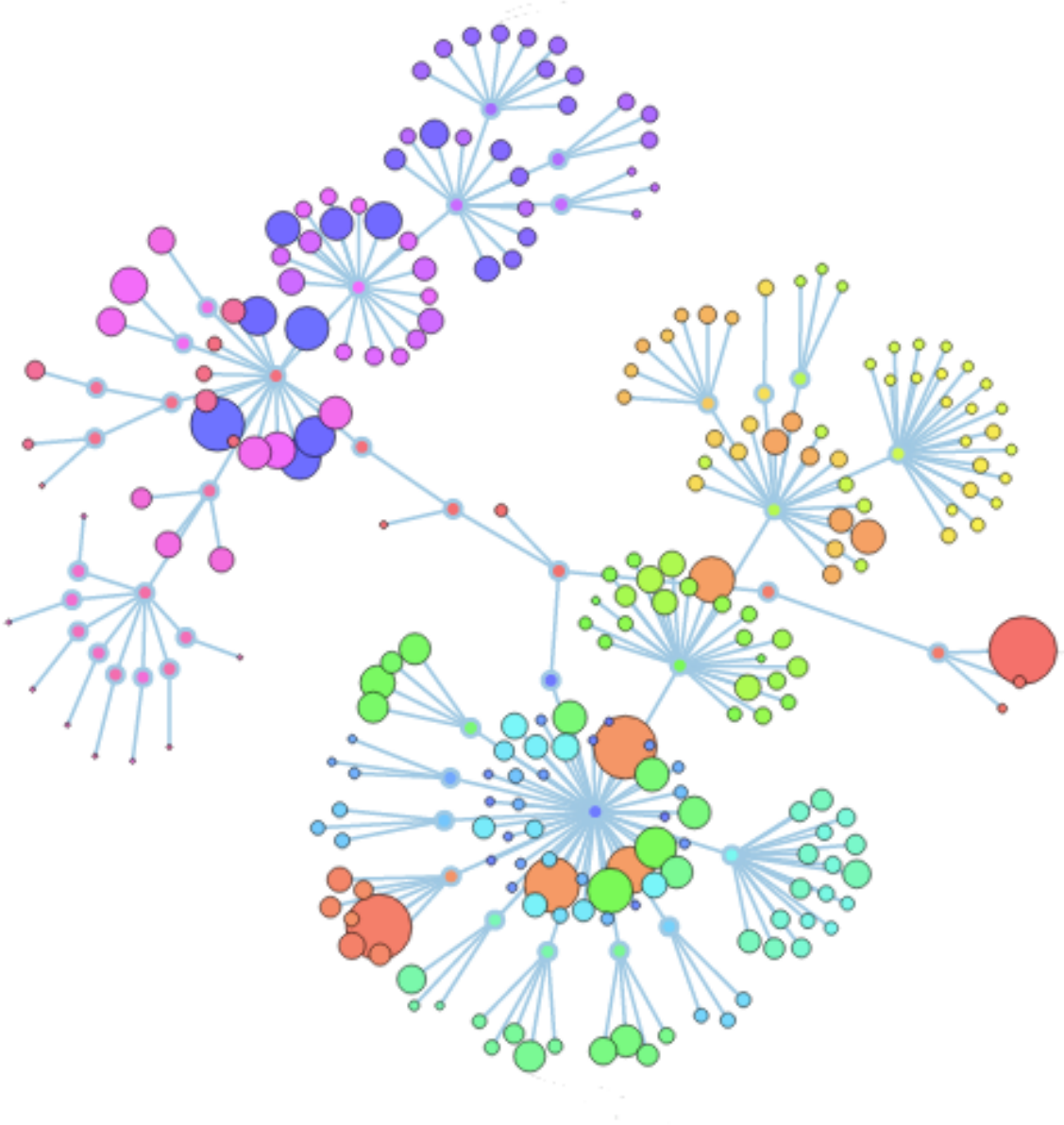

Рис. 3.4: приклад дерева, побудованого з допомогою *CodeFlower*

та аналізу було використано бібліотеку *CodeMirror*.

CodeMirror - це потужна бібліотека для підсвічування синтаксису близько сорока з гаком мов програмування і розмітки. Головною відмінністю від досить відомого *Syntax Highlighter* є реалізація динамічного просунутого підсвічування коду буквально на ходу, як це приміром реалізовано в настільних додатках типу *Geany* або *Notepad++*. Крім усього іншого присутня налаштування клавіш швидкого доступу, показ ключових слів по *Ctrl+Space*, виділення поточної рядки (де зупинився курсор мишки), напівавтоматична розстановка відступів і багато багато іншого.

CodeMirror є внутрішньо-браузерним *javascript*-компонентом, який був

розроблений *Marijn Haverbeke*. *CodeMirror* сумісний з *Firefox* 3 або вище, *Google Chrome, Safari* 5.2 або вище, *Opera* 9 або вище, а також *Internet Explorer* 8 або вище в стандартному режимі.

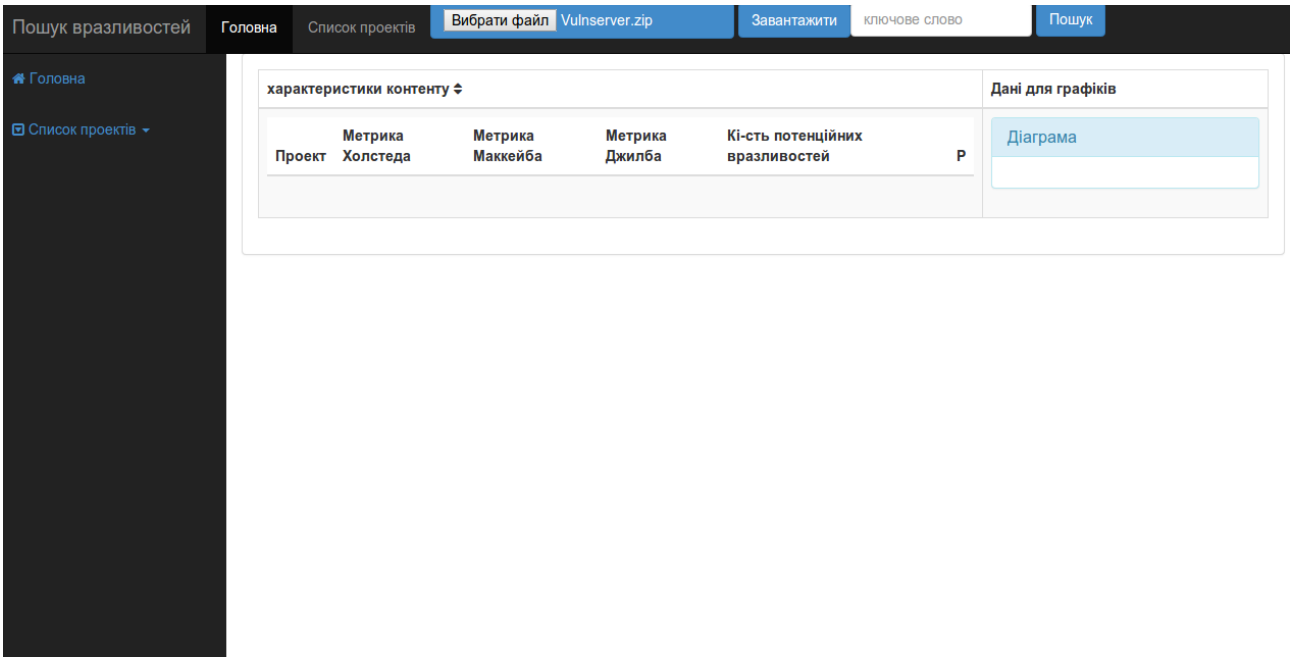

Рис. 3.5: головна сторінка

Загалом, інтерфейс користувача є доволі простим та інтуєтивним (Рис. 3.5).

Для підготовки проекту до аналізу необхідно зробитти його зжаття та архівування в формат *zip*. Після вивантаження зжатого архіву на сервер відбувається його розархівування з наступним аналізом всіх файлів, які він містить на предмет вразливостей та обчислення вищенаведених метрик його коду, з занесенням в БД відповідних результатів аналізу.

Після даної операції існує можливість переглянути його властивості, зробити висновок щодо організації подальшого досліджнення цільових програм для реалізації кібернетичного впливу.

На сторінці списку завантажених(проаналізованих) проектів виводиться список проктів, елементами якого є пара - таблиця та відповідна діаграма, яка відображає властивості проекту.

Окрім цього можна також скористатись навігацією по вихідним текстам проекту, де також відображається інформація про метрики коду та кількість потенційних вразливостей для кожного файлу проекту (Рис. 3.6).

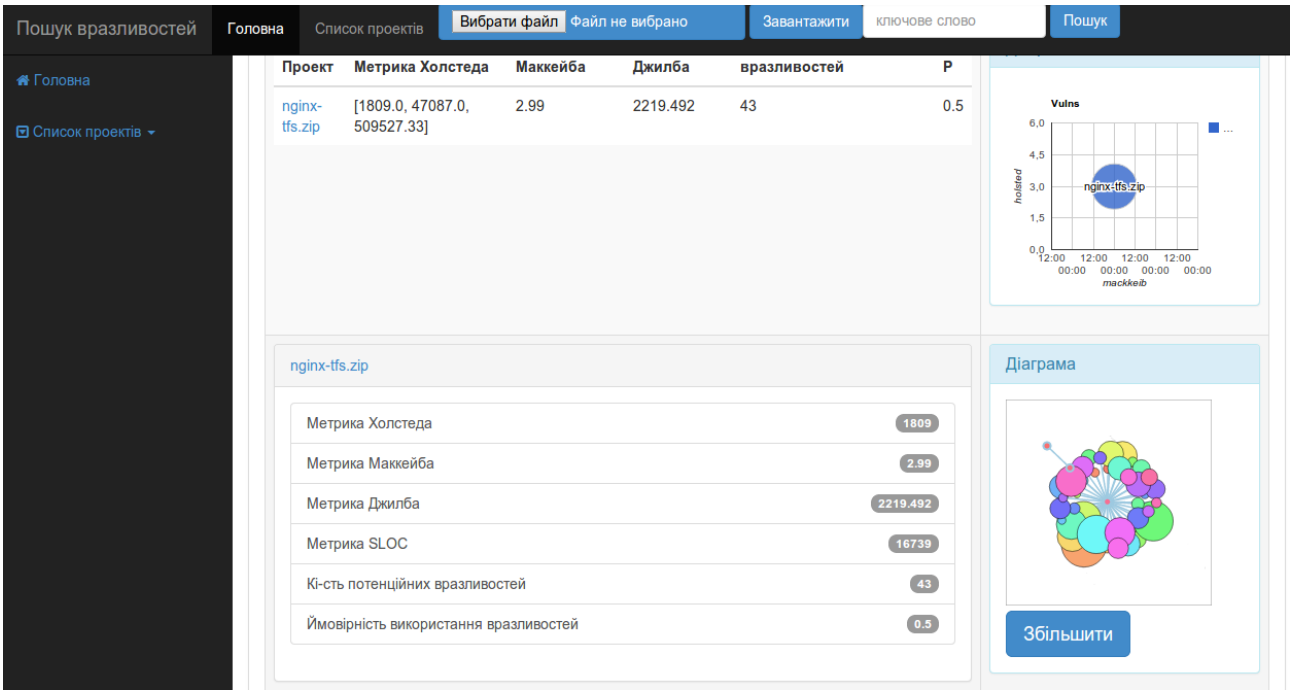

Рис. 3.6: властивості проекту

3.3 Керівництво щодо розгортання та експлуатації

Для встановлення та налаштування роботи програмного модулю необхідно встановити наступні складові:

1. встановити інтерпретатор *Python 2.7*;
2. встановити менеджер пакетів *Python - PIP*;
3. завантажити з офіційного сайту та встановити пакет інструментальних засобів *Google App Engine SDK*;
4. за допомогою менеджеру пакетів встановити бібліотеку лінійної алгебри *NumPy*;
5. завантажити *Google App Engine Launcher* - програмний засіб управління *Google App Engine*;
6. запустити програмний модуль за допомогою *Google App Engine Launcher*.
7. за допомогою браузера здійснити доступ до веб-сервісі, по налаштованому порту.

Розглянемо дещо детальніше даний процес для операційної системи Linux:

встановимо *Python*

```
$ sudo apt-get install python27
```

встановимо пекетний менеджер *Python*

```
$ sudo apt-get install python-pip27
```

встановимо *Google App Engine SDK*

```
$ cd /tmp
$ wget http://googleappengine.googlecode.com/files/google_appengine_1.8.9.z
$ unzip google_appengine_1.8.9.zip
$ cd google_appengine
$ sudo python setup.py build install
```

за допомогою менеджеру пакетів встановимо бібліотеку лінійної алгебри *NumPy*

```
$ sudo pip install numpy
```

завантажимо *Google App Engine Launcher* - програмний засіб управління *Google App Engine*

```
$ cd /opt/
$ svn checkout http://google-appengine-wx-launcher.googlecode.com/svn/trunk
$ cd google-appengine-wx-launcher-read-only
$ python GoogleAppEngineLauncher.py
```

запустимо програмний модуль за допомогою *Google App Engine Launcher*

```
$ dev_appserver.py <path to project.module>
```

Висновки

Отже, підводячи підсумок практичної реалізації можна зазначити, що мета оцінки та програмного коду для ефективної організації аналізу проектів на предмет можливості використання їх потенційно-небезпечних вразливостей досягнута. Вищеперераховані програмні методи можуть бути використані в подальшому в мережах спеціального призначення.

4. ОЦІНКА ОБЧИСЛЮВАЛЬНОЇ РОБОТИ ЗАСТОСУВАННЯ ПРОГРАМНОГО МОДУЛЮ ПРИ ВИКОНАННІ ТИПОВИХ ЗАВДАННЬ ДОСЛІДЖЕННЯ ПОТЕНЦІЙНО-НЕБЕЗПЕЧНИХ ДЕФЕКТІВ ЦІЛЬОВИХ ПРОГРАМ

Програмним продуктом, який буде досліджено, буде один із модулів веб-серверу *nginx*, а саме, *TFS*.

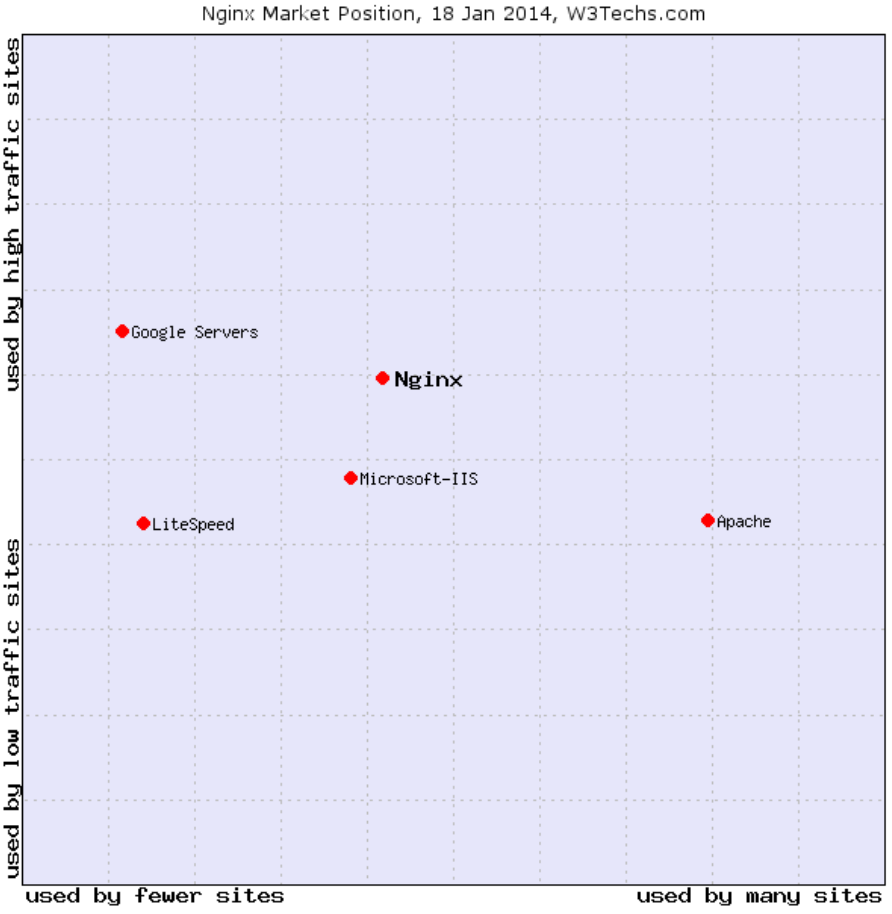

Рис. 4.1: статистика використання *nginx* за 18 січня 2014 року згідно *W3Techs*

nginx — це вільний веб-сервер і проксі-сервер. Є версії для сімейства *Unix*-подібних операційних систем *(FreeBSD, GNU/Linux, Solaris, Mac OS X)* та *Microsoft Windows*.

Згідно з квітневим 2012 року звітом компанії *Netcraft nginx* використовується на 12.76% всіх активних сайтів і на 10.09% з мільйона найвідвідуваніших сайтів у світі. За рік до того *nginx* використовувався на 8.68% всіх активних сайтів і 6.52% популярних сайтів. За рік *nginx* переступив десятивідсоткову

межу і витіснив *IIS* на третє місце в рейтингу популярності активних сайтів. Звіт налічує близько 23.4 млн хостів під управлінням *nginx*. За даними *W3Techs* на 18 січня 2014 11% з мільйона найвідвідуваніших сайтів у світі використовують *nginx*, тоді як у квітні 2011 року цей показник становив 6.8%. У Росії *nginx* використовується на 58.2% найбільш відвідуваних сайтів (рік тому — 46.9%) (Рис. 4.1).

TFS це розподілена файлова система, розроблена *Taobao*. Метою розподіленої файлової системи з високою доступністю, високою продуктивністю і низькою вартістю, *TFS*-це *linux*-файлова система, яка забезпечує високу надійність і паралельного доступу за рахунок дублювання, копіювання і технології балансування навантаження. *TFS*, в основному, призначений для невеликих файлів (менше 1 Мб. Він приймає плоску структуру замість традиційної структури каталогів. *TFS* буде генерувати 18 байт довжини файлу після збереження даних для завантаження файлів. Користувачі можуть отримувати доступ до своїх даних з унікальним ім'ям.

4.1 Опис випробування (натурного)

Ефективність програмного забезпечення — характеристика програмного забезпечення, ступінь відповідності ПМ до вимог. При цьому вимоги можуть трактуватись по-різному, що породжує декілька незалежних визначень терміну. Частіше за все, використовують визначення *ISO 9001*, згідно з яким якість – це "ступінь відповідності наявних характеристик вимогам".

Фактори ефективності — це нефункціональні вимоги до ПМ, що відносяться до, наприклад, надійності та продуктивності програм.

Для оцінки ефективності розробленого програмного забезпечення використано такі основні фактори:

- Зрозумілість - Призначення ПМ повинно бути зрозумілим з самої програми та документації.
- Повнота - Всі необхідні частини програми повинні бути представлені та реалізовані.
- Стислість - Відсутність надлишкової інформації та такої, що дублюється.

- Можливість портування - Легкість в адаптації програми до інших умов: архітектури, платформи, операційної системи тощо.
- Узгодженість - Вся документація та код повинні виконуватися за єдиними угодами, використовувати єдині формати та позначення
- Зручність використання - Простота та зручність використання програми. Ця вимога відноситься в першу чергу до інтерфейсу користувача.
- Надійність - Відсутність відмов та збоїв у роботі програми, а також простота виправлення помилок.
- Ефективність - Наскільки раціонально програма відноситься до ресурсів (пам'ять, процесор) при виконанні своїх задач.
- Безпечність.

Зрозумілість

Призначення розробленого програмного модулю – полегшити роботу при аналізі програмних продуктів з метою використання їх вразливостей, та забезпечити досягнення мети у повному обсязі. Призначення являється зрозумілим, а документація користувача надає вичерпні відповіді щодо роботи програмного модулю, завдяки чому суть його призначення можна вважати доступним.

Повнота

Програмний продукт являє собою першу версію програмного додатка для аналізу вихідних текстів програмних продуктів на предмет наявності доступних у використанні потенційно-небезпечних вразливостей. Усі функції, покладені на дане програмне забезпечення реалізовані у повному обсязі згідно до зазначених заздалегідь вимог, та працюють коректно.

Стислість

Програмний продукт був неодноразово перевірений і протестований на наявність зайвих та дубльованих процедур та функцій. У результаті перевірок надлишкової та дубльованої інформації виявлено не було.

Можливість портування

Даний програмний продукт був реалізований за допомогою мови програмування *Python*, що надає змогу користувачу застосовувати його на більшості поширених операційних систем, на які портовано інтерпретатор *Python*.

Узгодженість

Призначення розробленого програмного модулю – полегшити роботу при аналізі програмних продуктів з метою використання їх вразливостей, та забезпечити досягнення мети у повному обсязі. Призначення являється зрозумілим, а документація користувача надає вичерпні відповіді щодо роботи програмного модулю, завдяки чому суть його призначення можна вважати доступним.

Зручність використання

Питання зручності використання створеного програмного продукту задовольняється наявністю легкого в експлуатації інтерфейсу користувача, наявністю максимально зрозумілої довідки користувача, а також легкості у супроводженні з точки зору користувача-програміста. Однією з висунутих заздалегідь вимог до програмного продукту була відкритість коду. Ця вимога реалізована, і надає можливість у будь-який час за потреби внести власні зміни до версії програмного продукту. Інтерфейс користувача реалізований таким чином, щоб максимально забезпечити зручність у застосуванні. Він є інтуїтивно зрозумілим для будь-якого користувача з базовими знаннями по роботі з ЕОМ.

Безпечність

Обробка інформації під час застосування здійснюється на серверній частині програмного продукту, а не на сторонньому сервері. Це виключає можливість потрапляння інформації стороннім особам, і являється суттєвим внеском у питання безпеки.

Оцінка отриманих результатів.

В якості показника для цього виду оцінки використовують:

1. Скорочення часу вирішення задачі при використання ЕОМ (T,r) обраховується за формулою:

$$T = \frac{Tp - Ta}{N} \tag{4.1}$$

де:

Тр – час вирішення задачі без ЕОМ;

Та – час вирішення задачі на ЕОМ;

N – частота вирішення задачі.

2. Відповідно до формули (4.1) Тр обраховується за формулою:

$$Tp = \text{Тпр} + \text{Трр} + \text{Тор} \tag{4.2}$$

де:

Тпр – час збору і підготовки вихідних даних для вирішення задачі вручну (40 хв.);

Трр – час вирішення задачі (15 хв);

Тор – час на оформлення результатів вирішення (5 хв).

В нашому випадку Тр=60 хв.

3. Відповідно до формули (4.1) Та обраховується за формулою:

$$Ta = \text{Тпа} + \text{Тра} + \text{Тоа} \tag{4.3}$$

де:

Тпа – час збору інформації для вводу в ЕОМ (5 с);

Тра – час вирішення задачі на ЕОМ (2 с);

Тоа – час обробки і оформлення результату (0.3 с).

В нашому випадку Та=7,5 с.

Коефіцієнт відносної економії часу при вирішенні задачі на ЕОМ (КОСТ,%) обраховується за формулою:

$$KOCT = \frac{Tp - Ta}{Tp}100\% = \frac{T}{N * Tp}100\% \quad (4.4)$$

В нашому випадку КОСТ=99.78%. Показник ефективності по часу вирішення зростає зі збільшенням об'єму інформації, яка обробляється та частоти вирішення задачі.

Надійність

Програмний продукт був повність протестований на наявність помилок та збоїв при роботі. Після ряду тестів та пробного практичного застосування було відлагоджено усі наявні недоліки. Окрім цього було розроблено ряд *unit*-тестів, що дозволило частково автоматизувати процес тестування та звільнитись від більшості ручних операцій.

Ефективність

Задля оцінки ефективності було проведено наступний експеримент: на вхід аналізатора було подано файл вихідного коду програми, який містив вразливості переповнення буфера різних типів : переповнення стеку, кучі, переповнення керуючих змінних. На виході було отримано перелік потенційно вразливих елементів проекту.

4.2 Порівняльний аналіз результатів

Після завантаження досліджуваного модулю на сервер, було проведено аналіз і отримано наступні результати:

назва файлу	SLOC	V	Z(G)	Rup	vulns	P	p
rc_server_message.c	818	2852.0	2.61	168.75	1	0.15	0.50
common.c	834	2376.0	1.29	214.29	1	0.12	0.50
json.c	345	1104.0	2.08	67.54	1	0.13	0.50
module.c	1246	3422.0	1.7	267.28	1	0.13	0.50
raw_fsname.c	180	1083.0	1.86	79.43	1	0.14	0.50
local_block_cache.c	443	1506.0	2.26	83.84	1	0.13	0.50
peer_connection.c	464	1589.0	3.67	96.24	1	0.22	0.50
timers.c	185	522.0	1.87	41.86	1	0.15	0.50
block_cache.c	163	464.0	1.32	32.58	1	0.09	0.50
common.h	286	580.0	1.56	119.03	1	0.32	0.50
duplicate.c	546	1597.0	1.78	106.01	1	0.12	0.50
name_server_message.c	873	2971.0	2.87	166.11	1	0.16	0.50
root_server_message.c	71	208.0	1.58	28.52	1	0.22	0.50
server_handler.c	1490	4235.0	2.69	257.18	1	0.16	0.50
restful.c	754	2370.0	3.68	102.10	1	0.16	0.50
connection_pool.c	322	924.0	1.82	66.21	1	0.13	0.50
data_server_message.c	1633	5848.0	3.12	265.81	1	0.14	0.50
rc_server_info.c	324	1032.0	1.86	73.08	1	0.13	0.50
tfs.c	2281	7339.0	2.46	449.20	1	0.15	0.50
meta_server_message.c	829	3058.0	2.78	179.36	1	0.16	0.50
remote_block_cache.c	484	1465.0	1.84	94.28	1	0.12	0.50
tair_helper.c	187	542.0	1.33	28.29	1	0.07	0.50

Як видно з таблиці, даний програмний модуль містить ряд помилок, і ймовірність їх використання є середньою - для більш точних даних необхідна більша кількість досліджуваних проектів та експертних оцінок фахівців даної області.

Висновки

Даний програмний комплекс успішно справляється з задачами дослідження та пошуку потенційно-небезпечних дефектів реакції програм, збереженням та візуалізацією даних аналізу. Він є простим та інтуїтивним в використанні, не потребує глибоких знань та навичок аналізу коду а тому є придатним для швидкого ознайомлення з інтегрованими властивостями досліджуваного програмного продукту.

ЗАКЛЮЧЕННЯ

Підводячи загальний підсумок роботи необхідно зазначити.

По-перше, незважаючи на стрімкий розвиток інформаційних технологій, зокрема мов програмування, інструментів та засобів реалізації програмних продуктів використання низькорівневих мов програмування з відкритим управлінням памяттю складає значну долю в порівнянні з високорівневими мовами. Як відомо, саме ці мови і мають недоліки, які полягають у можливості модифікації поведінки зі сторони.

По-друге, на сьогодні існує низка інструментів аналізу вихідних текстів програм, яка дозволяє отримати багато корисної інформації про вразливі місця в коді, але ці засоби, в основному, являються не повними, або частіше за все, не дають повної картини, яка б дозволила в короткий термін прийняти рішення щодо спрямування роботи над тим чи іншим модулем.

По-третє, на основі проведених експертних сучасних метрик програмного забезпечення перевага була віддана метрикам Холстеда, Маккейб та Джилба, котрий складає строгого математичний алгоритм, а також запропоновано використання змішаної оцінки виходячи з оцінок цих метрик та кількісті потенційно-небезпечних вразливостей вихідних тектстів програм для підвищення точності при організації кібернетичного впливу. Дана модель надає можливість на основі екстраполяції попередніх результатів давати оцінку успішності проведення кібернетичного впливу з використанням тієї чи іншої вразливостей.

По-четверте, розглянувши сучасні тенденції кібернетичного впливу, та зробивши аналіз поширених вразливостей програмного забезпечення, способи їх використання та засоби виявлення, можна зробити висновок що дана предметна область поки що мало досліджена та потребує більш ґрунтовного вивчення.

Запропонована модель забезпечує в короткий термін проведення оцінки коду, при цьому результати оцінки можуть бути зкореговані спеціалістом данної предметної області в процесі роботи, що дозволить в подальшому аналізі отримувати більш точні прогнози.

Запропоновано практичну реалізацію метою якої є забезпечення зручного та швидкого аналізу програмних продуктів для подальшої, ефективної роботи з ними.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

[1] *МХ.Холстед.* Начало науки о программах. — 1981.

[2] *А.А.Малюк.* Информационная безопасность концептуальные и методологические основы защиты информации. — 2004.

[3] *А.В.Альфред Сети Равви* Компиляторы: принципы, технологии и инструменты. — Издательский дом “Вильямс”, 2003. — Р. 768.

[4] *А.Федоров* Windows Azure: облачная платформа Microsoft - Microsoft. — Microsoft, 2010.

[5] *В.А.Шеховцов.* Операційні системи. — Харків, 2010.

[6] *В.И.Шелехов.* Структура программы в языково-ориентированном потоковом анализе. — Новосибирск: Наука, 1998.

[7] *В.П.Мельников.* Информационная безопасность и защита информации. — 2009.

[8] *Г.В. Курячий* Операционная система Linux. — Москва: "INTUIT.ру 2003.

[9] *Г.Г.Почепцов* . Информационные войны. — Ваклер, 2000. — Р. 576.

[10] *Г.И.Кайгородцев.* Введение в курс метрической теории и метрологии программ / Ed. by 2011. — Новосибирск: НГТУ.

[11] *Д.Бэкон* Операционные системы. — Киев: Издательская группа BHV, 2004.

[12] *Д.С.Фостер.* Защита от взлома. Сокеты, shell-код, эксплойты. — Издательский дом “Вильямс”, 2008. — Р. 784.

[13] *Д.Эриксон.* Хакинг: искусство эксплойта. — Санкт-Петербург — Москва, 2005.

[14] *С.К. Черноножкин.* Меры сложности программ. — Новосибирск: Наука, 2009.

[15] *У.Себеста-Роберт.* Основные концепции языков программирования. — Издательский дом “Вильямс”, 2001. — P. 672.

[16] *Э.Д.Хопкрофт Мотвани Раджив* Введение в теорию автоматов, языков и вычислений. — Издательский дом “Вильямс”, 2002. — P. 528.

[17] *Э.Таненбаум.* Современные операционные системы. — 2-е издание, СПб. Питер, 2005.

ДОДАТКИ

```
class Metrix(db.Model):
    mackkeib = db.StringProperty()
    holsted = db.StringProperty()
    jilb = db.StringProperty()
    sloc = db.StringProperty()


class Vulnerability(db.Model):
    vulnerability = db.StringProperty()


class Project(db.Model):
    short = db.StringProperty()
    name = db.StringProperty()
    metrix = db.ReferenceProperty(Metrix, default=None)
    vulnerability = db.ReferenceProperty(Vulnerability, default=None)
    potential = db.FloatProperty()
    p = db.FloatProperty()


class SourceFile(db.Model):
    short = db.StringProperty()
    project = db.ReferenceProperty(Project)
    name = db.StringProperty()
    source = db.BlobProperty()
    metrix = db.ReferenceProperty(Metrix, default=None)
    vulnerability = db.ReferenceProperty(Vulnerability, default=None)
    potential = db.FloatProperty()
    p = db.FloatProperty()


__author__ = 'andrew.vasyltsiv'


from collections import namedtuple
from StringIO import StringIO
from md5 import md5
```

```
import re
from functools import reduce
from interpolation import LinearInterpolation
Source = namedtuple("Source", "project file_name file_source file_db_item h

class SourceFilesFormatError(Exception):
    pass

class SourceFileFormatError(SourceFilesFormatError):
    pass

from app.models import (
        Project,
        SourceFile,
        Metrix,
        Vulnerability
    )

from vertix.metrix import (
        get_holsted,
        get_mackkeib,
        get_jilb,
        get_sloc
    )

from vertix.vulns import (
        get_vulns_count
    )

from vertix import get_ast_from_text

def remove_Directives(source):
    source = re.sub(r"/\*([^*])+.+[^/]/", "", source)

    source = re.sub(r"/\*((?<=\*)[^/]+)/", "", source)
    file = StringIO(source)
    result = []
    for line in file.readlines():
        if line.lstrip(' ').startswith('#'):
            continue
        if line.lstrip(' ').startswith('//'):
```

```
            continue

        if line.find('//')>0:
            line = line.rpartition('//')[0]
        result.append(line)
    return '\n'.join(result)


class SourceProcessor():
    def __init__(self, project_files, project_name):
        print project_name
        if not isinstance(project_files, dict):
            raise TypeError ("project_files have to be dict")
        if not isinstance(project_name, str):
            raise TypeError("project_name must to be str")

        if project_files['error'] == False:
            if isinstance (project_files['project'], (list)):

                self.project_files = project_files['project']
                self.project_name = project_name
                self.project = Project.get_or_insert('name='+self.project_na
                self.project.name = self.project_name
                self.project.short = md5(project_name).hexdigest()
                self.init_sources_for_extrapolation()
                self.files = []
                self.process_sources()
                self.project.put()
            else:
                raise SourceFilesFormatError
        else:
            raise SourceFilesFormatError




    @staticmethod
    def calc_potential(v,GZ,Rup,vulns):
        if vulns == 0:
            return float((Rup*GZ)/float(v))
        else:
```

```
        return float((vulns*Rup*GZ)/float(v))

def init_sources_for_extrapolation(self):
    sources = SourceFile.all()
    tab = []
    tab_x = []
    tab_f = []
    for source in sources:
        if source.vulnerability is not None and \
            source.potential is not None and \
            source.p is not None:
            if int(source.vulnerability.vulnerability) >0 :
                tab.append((source.potential,source.p))
    tab = list(frozenset(tab))
    tab.sort()
    try:
        for i, item in enumerate(tab):
            if tab[i+1] == tab[i]:
                tab.pop(i)
    except IndexError:
        pass
    print tab
    self.tab_x = map(lambda x: x[0], tab)
    self.tab_f = map(lambda x: x[1], tab)



def calc_p(self,potential):
    if len(self.tab_x) < 2:
        return 0.5
    try:
        table = LinearInterpolation(
            x_index = tuple(self.tab_x),
            values = tuple(self.tab_f),
            extrapolate=True
            )
        return float(table(potential))
    except Exception, e:
        print e
        return 0.5
```

```
def process_sources(self):

    splitted_source = ""

    for file_name,file_source in self.project_files:

        ast_source = remove_Directives(file_source)


        try:

            ast = None
        except Exception, e:
            print e
            debuglines= 8
            line = int(str(e).split(':')[1])
            for i in range(debuglines):
                try:
                    print "INFO file[%s] line[%s]: %s" % (file_name,lin
                except:
                    pass
            ast = None
        if ast is not None:


            pass

        holsted, mackkeib,jilb, sloc = (
            function(file_source,ast) for function in (
                get_holsted,
                get_mackkeib,
                get_jilb,
                get_sloc
            )
        )


        if holsted != (-1,-1,-1):
            splitted_source += "\n"+file_source
```

```
vulns = get_vulns_count(file_source,ast)

metrix = Metrix(
                holsted = str(holsted),
                mackkeib = str(mackkeib),
                jilb = str(jilb),
                sloc = str(sloc)
            )
metrix.put()
vulnerability = Vulnerability(
                        vulnerability = str(vulns)
                    )
vulnerability.put()

potential = self.calc_potential(
                        holsted[0],
                        mackkeib,
                        jilb,
                        vulns
                    )
p = self.calc_p(potential)

short = self.project.short + md5(file_name).hexdigest()


print ("{} & {} & {} & {} & {} & {} & {} & {}\\\\".format(
        file_name,
        sloc,
        holsted[0],
        mackkeib,
        jilb,
        vulns,
        potential,
        p
    ))
source = Source(
        project = self.project,
        file_name = file_name,
        file_source = file_source,
        file_db_item = SourceFile(
                short = short,
```

```
                        project = self.project,
                        name = file_name,
                        source = file_source,
                        metrix = metrix,
                        vulnerability = vulnerability,
                        potential = potential,
                        p = p
                    ),
                holsted = holsted,
                mackkeib = mackkeib,
                jilb = jilb,
                sloc = sloc,
                vulns = vulns,
                potential = potential,
                p = p
            )
        source.file_db_item.put()
        self.files.append(source)
```

```
sloc = reduce(lambda x,y: x+y, map(lambda x: x.sloc, self.files))

holsted = get_holsted(splitted_source,None)

mackkeib = get_mackkeib(splitted_source,None)

jilb = get_jilb(splitted_source,None)

vulns = reduce(lambda x,y: (x+y), map(lambda x: x.vulns, self.files

potential = reduce(lambda x,y : x if x>y else y, map(lambda x: x.po

p = reduce(lambda x,y : x if x>y else y, map(lambda x: x.p, self.fi

self.project.potential = potential
self.project.p = p

metrix = Metrix(
```

```
                sloc = str(sloc),
                holsted = str(holsted),
                mackkeib = str(mackkeib),
                jilb = str(jilb)
            )
        metrix.put()
        self.project.metrix = metrix
        vulnerability = Vulnerability(
                vulnerability = str(vulns)
            )
        vulnerability.put()
        self.project.vulnerability = vulnerability


__author__ = 'andrew.vasyltsiv'




tokens = [

    'ID', 'TYPEID', 'INTEGER', 'FLOAT', 'STRING', 'CHARACTER',


    'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'MODULO',
    'OR', 'AND', 'NOT', 'XOR', 'LSHIFT', 'RSHIFT',
    'LOR', 'LAND', 'LNOT',
    'LT', 'LE', 'GT', 'GE', 'EQ', 'NE',


    'EQUALS', 'TIMESEQUAL', 'DIVEQUAL', 'MODEQUAL', 'PLUSEQUAL', 'MINUSEQUA
    'LSHIFTEQUAL','RSHIFTEQUAL', 'ANDEQUAL', 'XOREQUAL', 'OREQUAL',


    'INCREMENT', 'DECREMENT',
```

```
    'ARROW',


    'TERNARY',


    'LPAREN', 'RPAREN',
    'LBRACKET', 'RBRACKET',
    'LBRACE', 'RBRACE',
    'COMMA', 'PERIOD', 'SEMI', 'COLON',


    'ELLIPSIS',
]


t_PLUS             = r'\+'
t_MINUS            = r'-'
t_TIMES            = r'\*'
t_DIVIDE           = r'/'
t_MODULO           = r'%'
t_OR               = r'\|'
t_AND              = r'&'
t_NOT              = r'~'
t_XOR              = r'\^'
t_LSHIFT           = r'<<'
t_RSHIFT           = r'>>'
t_LOR              = r'\|\|'
t_LAND             = r'&&'
t_LNOT             = r'!'
t_LT               = r'<'
t_GT               = r'>'
t_LE               = r'<='
t_GE               = r'>='
t_EQ               = r'=='
t_NE               = r'!='


t_EQUALS           = r'='
t_TIMESEQUAL       = r'\*='
```

```
t_DIVEQUAL          = r'/='
t_MODEQUAL          = r'%='
t_PLUSEQUAL         = r'\+='
t_MINUSEQUAL        = r'-='
t_LSHIFTEQUAL       = r'<<='
t_RSHIFTEQUAL       = r'>>='
t_ANDEQUAL          = r'&='
t_OREQUAL           = r'\|='
t_XOREQUAL          = r'\^='


t_INCREMENT         = r'\+\+'
t_DECREMENT         = r'--'


t_ARROW             = r'->'


t_TERNARY           = r'\?'


t_LPAREN            = r'\('
t_RPAREN            = r'\)'
t_LBRACKET          = r'\['
t_RBRACKET          = r'\]'
t_LBRACE            = r'\{'
t_RBRACE            = r'\}'
t_COMMA             = r','
t_PERIOD            = r'\.'
t_SEMI              = r';'
t_COLON             = r':'
t_ELLIPSIS          = r'\.\.\.'


t_ID = r'[A-Za-z_][A-Za-z0-9_]*'


t_INTEGER = r'\d+([uU]|[lL]|[uU][lL]|[lL][uU])?'


t_FLOAT = r'((\d+)(\.\d+)(e(\+|-)?(\d+))? | (\d+)e(\+|-)?(\d+))([lL]|[fF])?
```

```
t_STRING = r'\"([^\\\n]|(\\.))*?\"'


t_CHARACTER = r'(L)?\'([^\\\n]|(\\.))*?\''


def t_COMMENT(t):
    r'/\*(.|\n)*?\*/'
    t.lexer.lineno += t.value.count('\n')
    return t


def t_CPPCOMMENT(t):
    r'//.*\n'
    t.lexer.lineno += 1
    return t
```

```
__version__     = "3.5"
__tabversion__ = "3.5"

import re, sys, types, copy, os, inspect


try:

    StringTypes = (types.StringType, types.UnicodeType)
except AttributeError:

    StringTypes = (str, bytes)




if sys.version_info[0] < 3:
    def func_code(f):
        return f.func_code
else:
    def func_code(f):
        return f.__code__


_is_identifier = re.compile(r'^[a-zA-Z0-9_]+$')




class LexError(Exception):
    def __init__(self,message,s):
         self.args = (message,)
```

```
        self.text = s


class LexToken(object):
    def __str__(self):
        return "LexToken(%s,%r,%d,%d)" % (self.type,self.value,self.lineno,
    def __repr__(self):
        return str(self)




class PlyLogger(object):
    def __init__(self,f):
        self.f = f
    def critical(self,msg,*args,**kwargs):
        self.f.write((msg % args) + "\n")

    def warning(self,msg,*args,**kwargs):
        self.f.write("WARNING: "+ (msg % args) + "\n")

    def error(self,msg,*args,**kwargs):
        self.f.write("ERROR: " + (msg % args) + "\n")

    info = critical
    debug = critical


class NullLogger(object):
    def __getattribute__(self,name):
        return self
    def __call__(self,*args,**kwargs):
        return self
```

```
class Lexer:
    def __init__(self):
        self.lexre = None



        self.lexretext = None
        self.lexstatere = {}
        self.lexstateretext = {}
        self.lexstaterenames = {}
        self.lexstate = "INITIAL"
        self.lexstatestack = []
        self.lexstateinfo = None
        self.lexstateignore = {}
        self.lexstateerrorf = {}
        self.lexreflags = 0
        self.lexdata = None
        self.lexpos = 0
        self.lexlen = 0
        self.lexerrorf = None
        self.lextokens = None
        self.lexignore = ""
        self.lexliterals = ""
        self.lexmodule = None
        self.lineno = 1
        self.lexoptimize = 0

    def clone(self,object=None):
        c = copy.copy(self)
```

```
    if object:
        newtab = { }
        for key, ritem in self.lexstatere.items():
            newre = []
            for cre, findex in ritem:
                 newfindex = []
                 for f in findex:
                      if not f or not f[0]:
                           newfindex.append(f)
                           continue
                      newfindex.append((getattr(object,f[0].__name__),f[
            newre.append((cre,newfindex))
            newtab[key] = newre
        c.lexstatere = newtab
        c.lexstateerrorf = { }
        for key, ef in self.lexstateerrorf.items():
            c.lexstateerrorf[key] = getattr(object,ef.__name__)
        c.lexmodule = object
    return c



def writetab(self,tabfile,outputdir=""):
    if isinstance(tabfile,types.ModuleType):
        return
    basetabfilename = tabfile.split(".")[-1]
    filename = os.path.join(outputdir,basetabfilename)+".py"
    tf = open(filename,"w")
    tf.write("# %s.py. This file automatically created by PLY (version
    tf.write("_tabversion   = %s\n" % repr(__tabversion__))
    tf.write("_lextokens    = %s\n" % repr(self.lextokens))
    tf.write("_lexreflags   = %s\n" % repr(self.lexreflags))
    tf.write("_lexliterals  = %s\n" % repr(self.lexliterals))
    tf.write("_lexstateinfo = %s\n" % repr(self.lexstateinfo))

    tabre = { }

    initial = self.lexstatere["INITIAL"]
    initialfuncs = []
```

```
    for part in initial:
        for f in part[1]:
            if f and f[0]:
                initialfuncs.append(f)

    for key, lre in self.lexstatere.items():
         titem = []
         for i in range(len(lre)):
              titem.append((self.lexstateretext[key][i],_funcs_to_names
         tabre[key] = titem

    tf.write("_lexstatere   = %s\n" % repr(tabre))
    tf.write("_lexstateignore = %s\n" % repr(self.lexstateignore))

    taberr = { }
    for key, ef in self.lexstateerrorf.items():
         if ef:
              taberr[key] = ef.__name__
         else:
              taberr[key] = None
    tf.write("_lexstateerrorf = %s\n" % repr(taberr))
    tf.close()




def readtab(self,tabfile,fdict):
    if isinstance(tabfile,types.ModuleType):
        lextab = tabfile
    else:
        if sys.version_info[0] < 3:
            exec("import %s as lextab" % tabfile)
        else:
            env = { }
            exec("import %s as lextab" % tabfile, env,env)
            lextab = env['lextab']

    if getattr(lextab,"_tabversion","0.0") != __tabversion__:
        raise ImportError("Inconsistent PLY version")

    self.lextokens      = lextab._lextokens
```

```
    self.lexreflags      = lextab._lexreflags
    self.lexliterals     = lextab._lexliterals
    self.lexstateinfo    = lextab._lexstateinfo
    self.lexstateignore = lextab._lexstateignore
    self.lexstatere      = { }
    self.lexstateretext = { }
    for key,lre in lextab._lexstatere.items():
         titem = []
         txtitem = []
         for i in range(len(lre)):
              titem.append((re.compile(lre[i][0],lextab._lexreflags | r
              txtitem.append(lre[i][0])
         self.lexstatere[key] = titem
         self.lexstateretext[key] = txtitem
    self.lexstateerrorf = { }
    for key,ef in lextab._lexstateerrorf.items():
         self.lexstateerrorf[key] = fdict[ef]
    self.begin('INITIAL')



def input(self,s):

    c = s[:1]
    if not isinstance(c,StringTypes):
        raise ValueError("Expected a string")
    self.lexdata = s
    self.lexpos = 0
    self.lexlen = len(s)



def begin(self,state):
    if not state in self.lexstatere:
        raise ValueError("Undefined state")
    self.lexre = self.lexstatere[state]
    self.lexretext = self.lexstateretext[state]
    self.lexignore = self.lexstateignore.get(state,"")
    self.lexerrorf = self.lexstateerrorf.get(state,None)
```

```
        self.lexstate = state



def push_state(self,state):
        self.lexstatestack.append(self.lexstate)
        self.begin(state)



def pop_state(self):
        self.begin(self.lexstatestack.pop())



def current_state(self):
        return self.lexstate



def skip(self,n):
        self.lexpos += n



def token(self):

        lexpos    = self.lexpos
        lexlen    = self.lexlen
        lexignore = self.lexignore
        lexdata   = self.lexdata
```

```
while lexpos < lexlen:

    if lexdata[lexpos] in lexignore:
        lexpos += 1
        continue


    for lexre,lexindexfunc in self.lexre:
        m = lexre.match(lexdata,lexpos)
        if not m: continue


        tok = LexToken()
        tok.value = m.group()
        tok.lineno = self.lineno
        tok.lexpos = lexpos

        i = m.lastindex
        func,tok.type = lexindexfunc[i]

        if not func:

            if tok.type:
                self.lexpos = m.end()
                return tok
            else:
                lexpos = m.end()
                break

        lexpos = m.end()



        tok.lexer = self
        self.lexmatch = m
        self.lexpos = lexpos

        newtok = func(tok)


        if not newtok:
```

```
            lexpos    = self.lexpos

            lexignore = self.lexignore

            break



        if not self.lexoptimize:
            if not newtok.type in self.lextokens:
                raise LexError("%s:%d: Rule '%s' returned an unknow
                    func_code(func).co_filename, func_code(func).co
                    func.__name__, newtok.type),lexdata[lexpos:])

        return newtok
    else:

        if lexdata[lexpos] in self.lexliterals:
            tok = LexToken()
            tok.value = lexdata[lexpos]
            tok.lineno = self.lineno
            tok.type = tok.value
            tok.lexpos = lexpos
            self.lexpos = lexpos + 1
            return tok


        if self.lexerrorf:
            tok = LexToken()
            tok.value = self.lexdata[lexpos:]
            tok.lineno = self.lineno
            tok.type = "error"
            tok.lexer = self
            tok.lexpos = lexpos
            self.lexpos = lexpos
            newtok = self.lexerrorf(tok)
            if lexpos == self.lexpos:

                raise LexError("Scanning error. Illegal character '
            lexpos = self.lexpos
            if not newtok: continue
```

```
                    return newtok

                self.lexpos = lexpos
                raise LexError("Illegal character '%s' at index %d" % (lexd

        self.lexpos = lexpos + 1
        if self.lexdata is None:
             raise RuntimeError("No input string given with input()")
        return None


    def __iter__(self):
        return self

    def next(self):
        t = self.token()
        if t is None:
            raise StopIteration
        return t

    __next__ = next
```

```
def _get_regex(func):
    return getattr(func,"regex",func.__doc__)
```

```
def get_caller_module_dict(levels):
    try:
        raise RuntimeError
    except RuntimeError:
        e,b,t = sys.exc_info()
        f = t.tb_frame
        while levels > 0:
            f = f.f_back
            levels -= 1
        ldict = f.f_globals.copy()
        if f.f_globals != f.f_locals:
            ldict.update(f.f_locals)

        return ldict
```

```
def _funcs_to_names(funclist,namelist):
    result = []
    for f,name in zip(funclist,namelist):
        if f and f[0]:
            result.append((name, f[1]))
        else:
            result.append(f)
    return result
```

```
def _names_to_funcs(namelist,fdict):
     result = []
     for n in namelist:
          if n and n[0]:
               result.append((fdict[n[0]],n[1]))
          else:
               result.append(n)
     return result
```

```
def _form_master_re(relist,reflags,ldict,toknames):
    if not relist: return []
    regex = "|".join(relist)
    try:
         lexre = re.compile(regex,re.VERBOSE | reflags)


         lexindexfunc = [ None ] * (max(lexre.groupindex.values())+1)
         lexindexnames = lexindexfunc[:]

         for f,i in lexre.groupindex.items():
              handle = ldict.get(f,None)
              if type(handle) in (types.FunctionType, types.MethodType):
                   lexindexfunc[i] = (handle,toknames[f])
                   lexindexnames[i] = f
              elif handle is not None:
                   lexindexnames[i] = f
                   if f.find("ignore_") > 0:
                        lexindexfunc[i] = (None,None)
                   else:
                        lexindexfunc[i] = (None, toknames[f])
```

```
            return [(lexre,lexindexfunc)],[regex],[lexindexnames]
        except Exception:
            m = int(len(relist)/2)
            if m == 0: m = 1
            llist, lre, lnames = _form_master_re(relist[:m],reflags,ldict,tokna
            rlist, rre, rnames = _form_master_re(relist[m:],reflags,ldict,tokna
            return llist+rlist, lre+rre, lnames+rnames
```

```
def _statetoken(s,names):
    nonstate = 1
    parts = s.split("_")
    for i in range(1,len(parts)):
         if not parts[i] in names and parts[i] != 'ANY': break
    if i > 1:
       states = tuple(parts[1:i])
    else:
       states = ('INITIAL',)

    if 'ANY' in states:
       states = tuple(names)

    tokenname = "_".join(parts[i:])
    return (states,tokenname)
```

```
class LexerReflect(object):
    def __init__(self,ldict,log=None,reflags=0):
        self.ldict      = ldict
        self.error_func = None
        self.tokens     = []
        self.reflags    = reflags
        self.stateinfo  = { 'INITIAL' : 'inclusive'}
        self.modules    = {}
        self.error      = 0

        if log is None:
            self.log = PlyLogger(sys.stderr)
        else:
            self.log = log


    def get_all(self):
        self.get_tokens()
        self.get_literals()
        self.get_states()
        self.get_rules()


    def validate_all(self):
        self.validate_tokens()
        self.validate_literals()
        self.validate_rules()
        return self.error


    def get_tokens(self):
        tokens = self.ldict.get("tokens",None)
        if not tokens:
            self.log.error("No token list is defined")
            self.error = 1
            return

        if not isinstance(tokens,(list, tuple)):
            self.log.error("tokens must be a list or tuple")
            self.error = 1
```

```
            return

        if not tokens:
            self.log.error("tokens is empty")
            self.error = 1
            return

        self.tokens = tokens


    def validate_tokens(self):
        terminals = {}
        for n in self.tokens:
            if not _is_identifier.match(n):
                self.log.error("Bad token name '%s'",n)
                self.error = 1
            if n in terminals:
                self.log.warning("Token '%s' multiply defined", n)
            terminals[n] = 1


    def get_literals(self):
        self.literals = self.ldict.get("literals","")
        if not self.literals:
            self.literals = ""


    def validate_literals(self):
        try:
            for c in self.literals:
                if not isinstance(c,StringTypes) or len(c) > 1:
                    self.log.error("Invalid literal %s. Must be a single ch
                    self.error = 1

        except TypeError:
            self.log.error("Invalid literals specification. literals must b
            self.error = 1

    def get_states(self):
        self.states = self.ldict.get("states",None)
```

```
    if self.states:
        if not isinstance(self.states,(tuple,list)):
            self.log.error("states must be defined as a tuple or list
            self.error = 1
        else:
            for s in self.states:
                if not isinstance(s,tuple) or len(s) != 2:
                    self.log.error("Invalid state specifier %s.
                    self.error = 1
                    continue
                name, statetype = s
                if not isinstance(name,StringTypes):
                    self.log.error("State name %s must be a stri
                    self.error = 1
                    continue
                if not (statetype == 'inclusive' or statetype == 'e
                    self.log.error("State type for state %s must
                    self.error = 1
                    continue
                if name in self.stateinfo:
                    self.log.error("State '%s' already defined",
                    self.error = 1
                    continue
                self.stateinfo[name] = statetype



def get_rules(self):
    tsymbols = [f for f in self.ldict if f[:2] == 't_' ]



    self.toknames = { }
    self.funcsym =  { }
    self.strsym =   { }
    self.ignore   = { }
    self.errorf   = { }

    for s in self.stateinfo:
        self.funcsym[s] = []
```

```
            self.strsym[s] = []

    if len(tsymbols) == 0:
        self.log.error("No rules of the form t_rulename are defined")
        self.error = 1
        return

    for f in tsymbols:
        t = self.ldict[f]
        states, tokname = _statetoken(f,self.stateinfo)
        self.toknames[f] = tokname

        if hasattr(t,"__call__"):
            if tokname == 'error':
                for s in states:
                    self.errorf[s] = t
            elif tokname == 'ignore':
                line = func_code(t).co_firstlineno
                file = func_code(t).co_filename
                self.log.error("%s:%d: Rule '%s' must be defined as a s
                self.error = 1
            else:
                for s in states:
                    self.funcsym[s].append((f,t))
        elif isinstance(t, StringTypes):
            if tokname == 'ignore':
                for s in states:
                    self.ignore[s] = t
                if "\\" in t:
                    self.log.warning("%s contains a literal backslash '

            elif tokname == 'error':
                self.log.error("Rule '%s' must be defined as a function
                self.error = 1
            else:
                for s in states:
                    self.strsym[s].append((f,t))
        else:
            self.log.error("%s not defined as a function or string", f)
            self.error = 1
```

```
        for f in self.funcsym.values():
            if sys.version_info[0] < 3:
                f.sort(lambda x,y: cmp(func_code(x[1]).co_firstlineno,func_
            else:

                f.sort(key=lambda x: func_code(x[1]).co_firstlineno)


        for s in self.strsym.values():
            if sys.version_info[0] < 3:
                s.sort(lambda x,y: (len(x[1]) < len(y[1])) - (len(x[1]) > l
            else:

                s.sort(key=lambda x: len(x[1]),reverse=True)


    def validate_rules(self):
        for state in self.stateinfo:




            for fname, f in self.funcsym[state]:
                line = func_code(f).co_firstlineno
                file = func_code(f).co_filename
                module = inspect.getmodule(f)
                self.modules[module] = 1

                tokname = self.toknames[fname]
                if isinstance(f, types.MethodType):
                    reqargs = 2
                else:
                    reqargs = 1
                nargs = func_code(f).co_argcount
                if nargs > reqargs:
                    self.log.error("%s:%d: Rule '%s' has too many arguments
                    self.error = 1
                    continue

                if nargs < reqargs:
```

```
            self.log.error("%s:%d: Rule '%s' requires an argument",
            self.error = 1
            continue

        if not _get_regex(f):
            self.log.error("%s:%d: No regular expression defined fo
            self.error = 1
            continue

        try:
            c = re.compile("(?P<%s>%s)" % (fname, _get_regex(f)), r
            if c.match(""):
                self.log.error("%s:%d: Regular expression for rule
                self.error = 1
        except re.error:
            _etype, e, _etrace = sys.exc_info()
            self.log.error("%s:%d: Invalid regular expression for r
            if '#' in _get_regex(f):
                self.log.error("%s:%d. Make sure '#' in rule '%s' i
            self.error = 1


    for name,r in self.strsym[state]:
        tokname = self.toknames[name]
        if tokname == 'error':
            self.log.error("Rule '%s' must be defined as a function
            self.error = 1
            continue

        if not tokname in self.tokens and tokname.find("ignore_") <
            self.log.error("Rule '%s' defined for an unspecified to
            self.error = 1
            continue

        try:
            c = re.compile("(?P<%s>%s)" % (name,r),re.VERBOSE | sel
            if (c.match("")):
                 self.log.error("Regular expression for rule '%s' m
                 self.error = 1
        except re.error:
            _etype, e, _etrace = sys.exc_info()
```

```
                self.log.error("Invalid regular expression for rule '%s
                if '#' in r:
                    self.log.error("Make sure '#' in rule '%s' is esca
                self.error = 1

        if not self.funcsym[state] and not self.strsym[state]:
            self.log.error("No rules defined for state '%s'",state)
            self.error = 1


        efunc = self.errorf.get(state,None)
        if efunc:
            f = efunc
            line = func_code(f).co_firstlineno
            file = func_code(f).co_filename
            module = inspect.getmodule(f)
            self.modules[module] = 1

            if isinstance(f, types.MethodType):
                reqargs = 2
            else:
                reqargs = 1
            nargs = func_code(f).co_argcount
            if nargs > reqargs:
                self.log.error("%s:%d: Rule '%s' has too many arguments
                self.error = 1

            if nargs < reqargs:
                self.log.error("%s:%d: Rule '%s' requires an argument",
                self.error = 1

    for module in self.modules:
        self.validate_module(module)
```

```
    def validate_module(self, module):
        lines, linen = inspect.getsourcelines(module)

        fre = re.compile(r'\s*def\s+(t_[a-zA-Z_0-9]*)\(')
        sre = re.compile(r'\s*(t_[a-zA-Z_0-9]*)\s*=')

        counthash = { }
        linen += 1
        for l in lines:
            m = fre.match(l)
            if not m:
                m = sre.match(l)
            if m:
                name = m.group(1)
                prev = counthash.get(name)
                if not prev:
                    counthash[name] = linen
                else:
                    filename = inspect.getsourcefile(module)
                    self.log.error("%s:%d: Rule %s redefined. Previously de
                    self.error = 1
            linen += 1
```

```
def lex(module=None,object=None,debug=0,optimize=0,lextab="lextab",reflags=
    global lexer
    ldict = None
    stateinfo  = { 'INITIAL' : 'inclusive'}
    lexobj = Lexer()
    lexobj.lexoptimize = optimize
    global token,input

    if errorlog is None:
        errorlog = PlyLogger(sys.stderr)
```

```
if debug:
    if debuglog is None:
        debuglog = PlyLogger(sys.stderr)


if object: module = object

if module:
    _items = [(k,getattr(module,k)) for k in dir(module)]
    ldict = dict(_items)
else:
    ldict = get_caller_module_dict(2)


linfo = LexerReflect(ldict,log=errorlog,reflags=reflags)
linfo.get_all()
if not optimize:
    if linfo.validate_all():
        raise SyntaxError("Can't build lexer")

if optimize and lextab:
    try:
        lexobj.readtab(lextab,ldict)
        token = lexobj.token
        input = lexobj.input
        lexer = lexobj
        return lexobj

    except ImportError:
        pass


if debug:
    debuglog.info("lex: tokens   = %r", linfo.tokens)
    debuglog.info("lex: literals = %r", linfo.literals)
    debuglog.info("lex: states   = %r", linfo.stateinfo)


lexobj.lextokens = { }
for n in linfo.tokens:
    lexobj.lextokens[n] = 1
```

```
if isinstance(linfo.literals,(list,tuple)):
    lexobj.lexliterals = type(linfo.literals[0])().join(linfo.literals)
else:
    lexobj.lexliterals = linfo.literals


stateinfo = linfo.stateinfo

regexs = { }

for state in stateinfo:
    regex_list = []


    for fname, f in linfo.funcsym[state]:
        line = func_code(f).co_firstlineno
        file = func_code(f).co_filename
        regex_list.append("(?P<%s>%s)" % (fname,_get_regex(f)))
        if debug:
            debuglog.info("lex: Adding rule %s -> '%s' (state '%s')",fn


    for name,r in linfo.strsym[state]:
        regex_list.append("(?P<%s>%s)" % (name,r))
        if debug:
            debuglog.info("lex: Adding rule %s -> '%s' (state '%s')",na

    regexs[state] = regex_list



if debug:
    debuglog.info("lex: ==== MASTER REGEXS FOLLOW ====")

for state in regexs:
    lexre, re_text, re_names = _form_master_re(regexs[state],reflags,ld
    lexobj.lexstatere[state] = lexre
    lexobj.lexstateretext[state] = re_text
    lexobj.lexstaterenames[state] = re_names
```

```
    if debug:
        for i in range(len(re_text)):
            debuglog.info("lex: state '%s' : regex[%d] = '%s'",state, i


for state,stype in stateinfo.items():
    if state != "INITIAL" and stype == 'inclusive':
         lexobj.lexstatere[state].extend(lexobj.lexstatere['INITIAL'])
         lexobj.lexstateretext[state].extend(lexobj.lexstateretext['INI
         lexobj.lexstaterenames[state].extend(lexobj.lexstaterenames['I

lexobj.lexstateinfo = stateinfo
lexobj.lexre = lexobj.lexstatere["INITIAL"]
lexobj.lexretext = lexobj.lexstateretext["INITIAL"]
lexobj.lexreflags = reflags


lexobj.lexstateignore = linfo.ignore
lexobj.lexignore = lexobj.lexstateignore.get("INITIAL","")


lexobj.lexstateerrorf = linfo.errorf
lexobj.lexerrorf = linfo.errorf.get("INITIAL",None)
if not lexobj.lexerrorf:
    errorlog.warning("No t_error rule is defined")


for s,stype in stateinfo.items():
    if stype == 'exclusive':
          if not s in linfo.errorf:
               errorlog.warning("No error rule is defined for exclusive
          if not s in linfo.ignore and lexobj.lexignore:
               errorlog.warning("No ignore rule is defined for exclusiv
    elif stype == 'inclusive':
          if not s in linfo.errorf:
               linfo.errorf[s] = linfo.errorf.get("INITIAL",None)
          if not s in linfo.ignore:
               linfo.ignore[s] = linfo.ignore.get("INITIAL","")


token = lexobj.token
```

```
    input = lexobj.input
    lexer = lexobj


    if lextab and optimize:
        lexobj.writetab(lextab,outputdir)

    return lexobj







def runmain(lexer=None,data=None):
    if not data:
        try:
            filename = sys.argv[1]
            f = open(filename)
            data = f.read()
            f.close()
        except IndexError:
            sys.stdout.write("Reading from standard input (type EOF to end)
            data = sys.stdin.read()

    if lexer:
        _input = lexer.input
    else:
        _input = input
    _input(data)
    if lexer:
        _token = lexer.token
    else:
        _token = token

    while 1:
        tok = _token()
        if not tok: break
        sys.stdout.write("(%s,%r,%d,%d)\n" % (tok.type, tok.value, tok.line
```

```
def TOKEN(r):
    def set_regex(f):
        if hasattr(r,"__call__"):
            f.regex = _get_regex(r)
        else:
            f.regex = r
        return f
    return set_regex


Token = TOKEN
```